Random Variate Generation with Formal Guarantees

Feras Saad and Wonyeol Lee

PLDI 2025

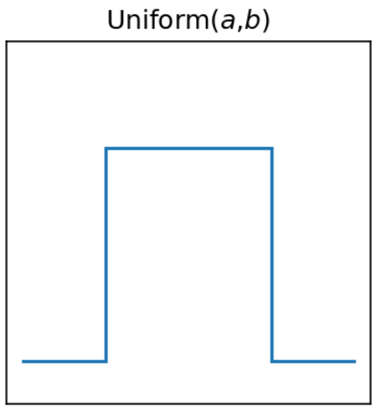
Seoul, Korea

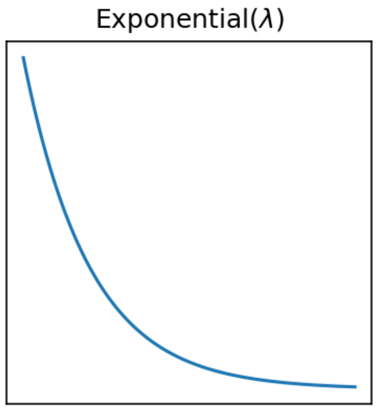
Agenda

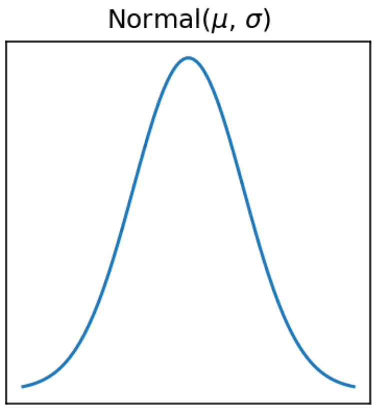
- Overview of Random Variate Generation
- Technical Approach
- Experimental Results
- Future Work

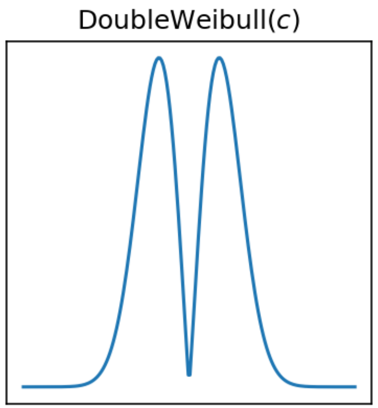
Probability distributions over the real line

Continuous

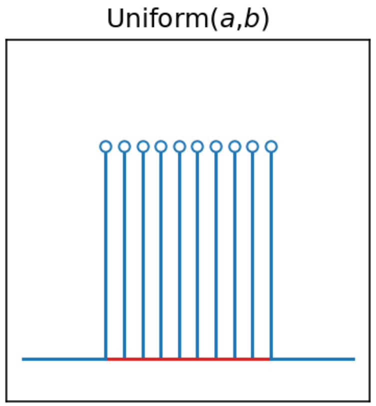


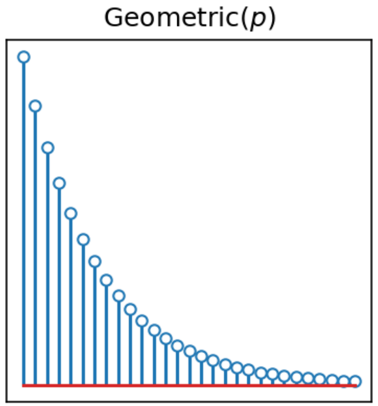


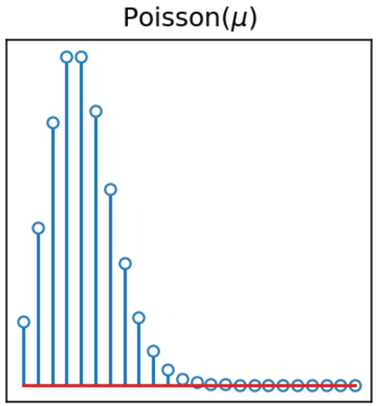


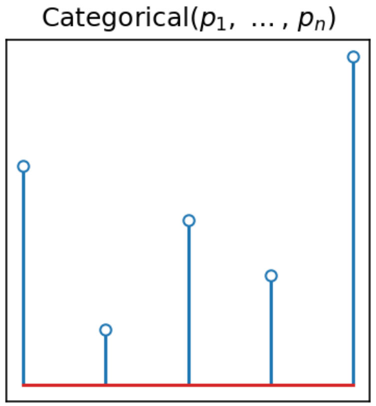


Discrete





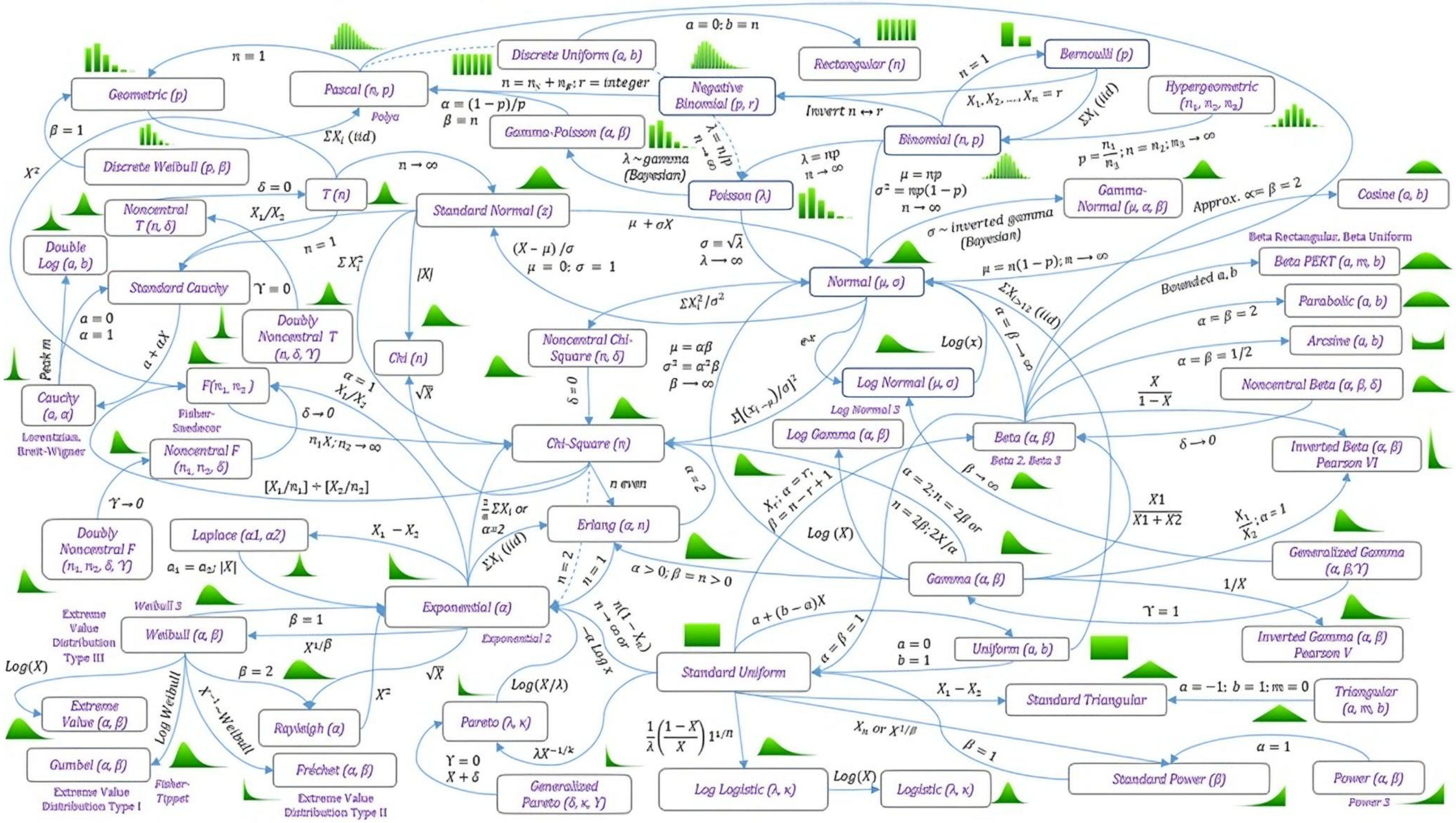




Mixed

```
if flip(0.5):
    return Poisson(7)
else:
    return Normal(0,1)
```

Probability distributions over the real line



https://rovusa.com/visual-guides-summaries-tables-models-list-and-useful-addenda/relationship-among-probability-distributions/ (reused with permission)

Probability distributions are central to many fields

• Robotics	*Probabilistic Robotics*	(Thrun et al. 2005)
• Computational Statistics	*Random Variate Generation*	(Devroye 1986)
• Operations Research	*Simulation Techniques in Operations Research*	(Harling 1958)
• Statistical Physics	*Monte Carlo Methods in Statistical Physics*	(Binder 1986)
• Financial Engineering	*Monte Carlo Methods in Financial Engineering*	(Glasserman 2003)
• Machine Learning	*An Intro to MCMC for Machine Learning*	(Andrieu+ 2003)
• Systems Biology	*Monte Carlo Methods in Biology*	(Manly 1991)
• Scientific Computing	*Monte Carlo Strategies in Scientific Computing*	(Liu 2001)
• Software Engineering	*Statistical Methods in Software Engineering*	(Singpurwalla+ 1999)
• Programming Languages	*Foundations of Probabilistic Programming*	(Barthe+ 2020)

we need reliable software abstractions and programming interfaces for interacting with probability distributions

Computing with probability distributions

A probability measure μ over $\mathbb{R}$ is a set function

$$\mu(A) \in [0,1] \qquad A \subset \mathbb{R} \text{ (measurable)}$$

<u>Operations of Interest</u>

- Generate a random variate $X \sim \mu$
- Compute cumulative probabilities $F(x) := \mu(-\infty, x]$ $(x \in \mathbb{R})$
- Compute survival probabilities $S(x) := \mu(x, \infty)$ $(x \in \mathbb{R})$
- Compute quantiles $Q(u) := \inf\{x \in \mathbb{R} \mid u \leq F(x)\}$ $(u \in [0,1])$

Computing with probability distributions

<u>Theorem</u>: The quantities μ, X, F, S, Q are all **mathematically equivalent** representations of a probability distribution over $\mathbb{R}$.

Examples of theoretical relationships

- $F(x) = \Pr(X \le x)$ "left-tail probability"
- $S(x) = \Pr(X > x)$ "right-tail probability"
- $S(x) = 1 - F(x)$ "additivity and normalization of measure"
- $X \stackrel{D}{=} Q(U), U \sim \mathrm{Uniform}(0,1)$ "inverse-transform theorem"
- $\mu(A) = \Pr\left(U \in X^{-1}(A)\right)$ "pushforward measure"

Computing with probability distributions

<u>Theorem</u>: The quantities μ, X, F, S, Q are all **mathematically equivalent** representations of a probability distribution over $\mathbb{R}$.

Does not hold in real-world software! ($\mathbb{R} \neq \mathbb{F}$)

SciPy

Methods

`rvs`(*args, **kwds)	Random variates of given type.
`cdf`(x, *args, **kwds)	Cumulative distribution function of the given RV.
`sf`(x, *args, **kwds)	Survival function (1 - `cdf`) at x of the given RV.
`ppf`(q, *args, **kwds)	Percent point function (inverse of `cdf`) at q of the given RV.

Computing with probability distributions

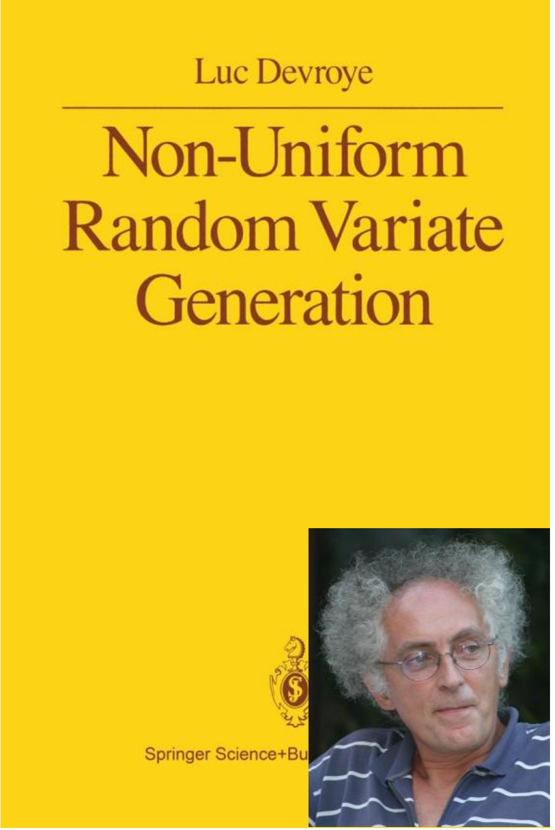
<u>Theorem</u>: The quantities μ, X, F, S, Q are all **mathematically equivalent** representations of a probability distribution over $\mathbb{R}$.

Does not hold in real-world software! ($\mathbb{R} \neq \mathbb{F}$)

Assumption 1. Our computer can store and manipulate real numbers.

Assumption 2. There exists a perfect uniform [0,1] random variate generator, i.e. a generator capable of producing a sequence $U_1, U_2, \ldots$ of independent random variables with a uniform distribution on [0,1].

Assumption 3. The fundamental operations in our computer include addition, multiplication, division, compare, truncate, move, generate a uniform random variate, exp, log, square root, arc tan, sin and cos. (This implies that each of these operations takes one unit of time regardless of the size of the operand(s). Also, the outcomes of the operations are real numbers.)



Failures of the real RAM model of computation

In the GNU Scientific Library, for the exponential distribution:

- The numerical CDF $F(x) = 1$ at $x \approx 17.33$ `-expm1(-x)`
- The random variate X can be as high as ≈ 22.18! `-log1p(-uniform())`

Failures of the real RAM model of computation

In the GNU Scientific Library, for the exponential distribution:

- The numerical CDF $F(x) = 1$ at $x \approx 17.33$ `-expm1(-x)`
- The random variate X can be as high as ≈ 22.18! `-log1p(-uniform())`

we found dozens of bug reports in widely used random variate libraries about such inconsistencies

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 issues/17007 np.random.geometric(10**-20) returns negative values

PyTorch issues/2257 CPU torch.exponential_function may generate 0 which can cause downstream NaN

Failures of the real RAM model of computation

In the GNU Scientific Library, for the exponential distribution:

- The numerical CDF $F(x) = 1$ at $x \approx 17.33$ `-expm1(-x)`
- The random variate X can be as high as ≈ 22.18! `-log1p(-uniform())`

we found dozens of bug reports in widely used random variate libraries about such inconsistencies

These problems also impact
- Differential privacy (Mironov 2012)
- Cryptography (Follath 2014)

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$\hat{8}$	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

Numerical approximations do not "commute"

Probability Measure μ

Random Variable Representations (i.e., Random Variate Generators) X_μ } equivalent

equivalent ↕

Cumulative Distribution Function Representation F_μ

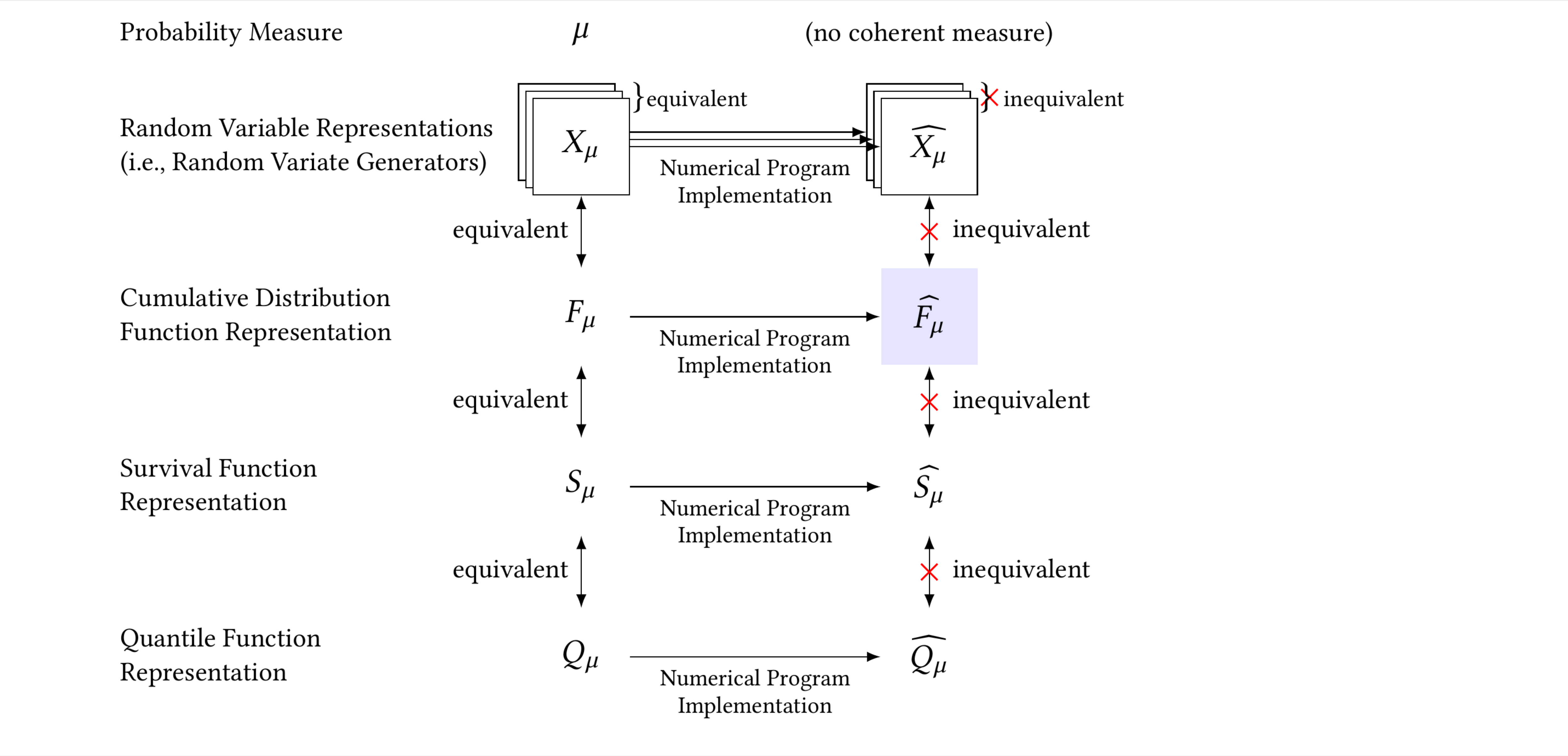
equivalent ↕

Survival Function Representation S_μ

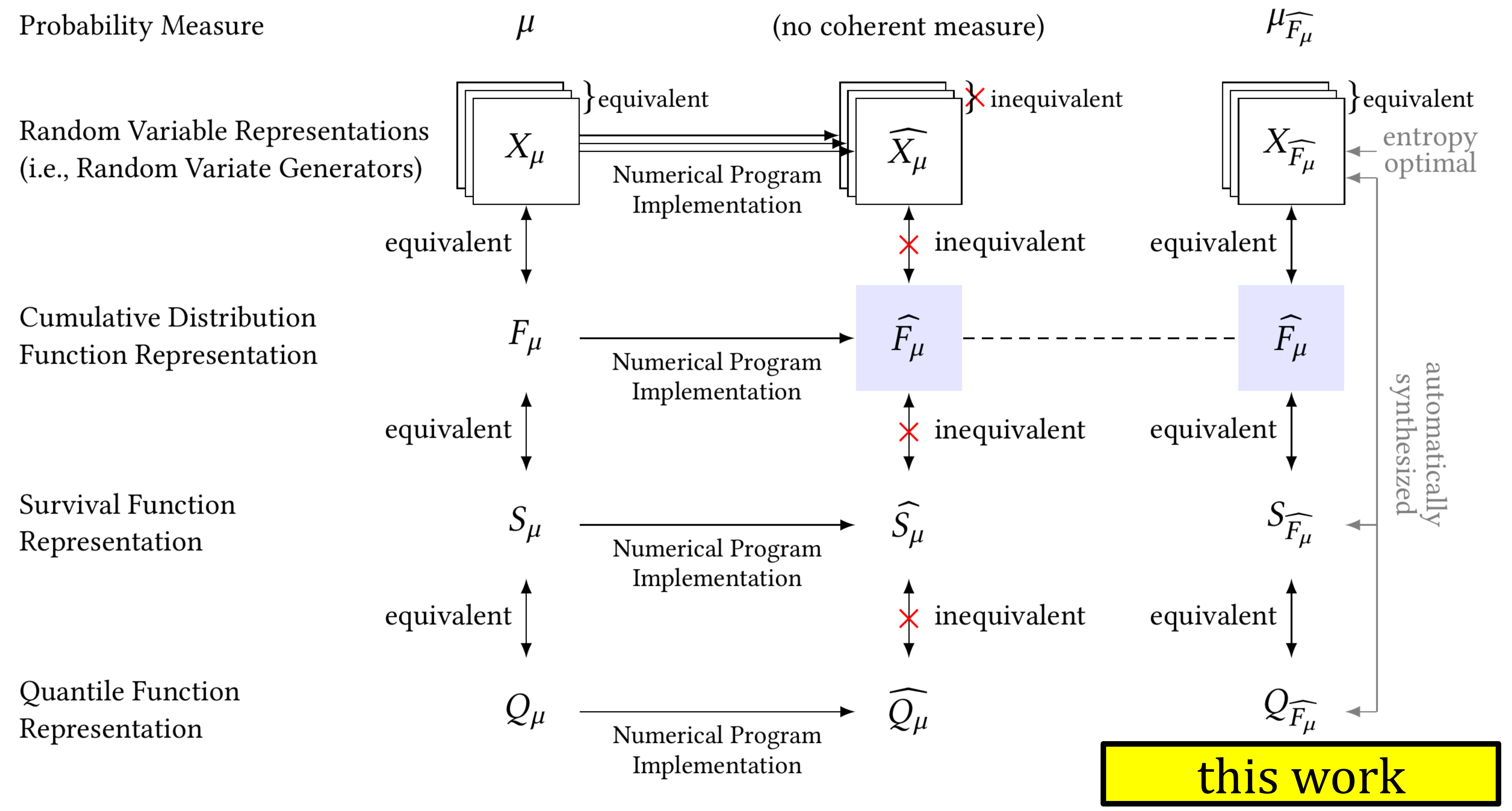
equivalent ↕

Quantile Function Representation Q_μ

Numerical approximations do not “commute”



This work: Automatically synthesizing generators



Example: Implementing a Gaussian distribution

We can also reuse CDF/SF implementations from existing libraries

```
// 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);
```

Example: Implementing a Gaussian distribution

We can also reuse CDF/SF implementations from existing libraries

```
// 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)
```

Example: Implementing a Gaussian distribution

We can also reuse CDF/SF implementations from existing libraries

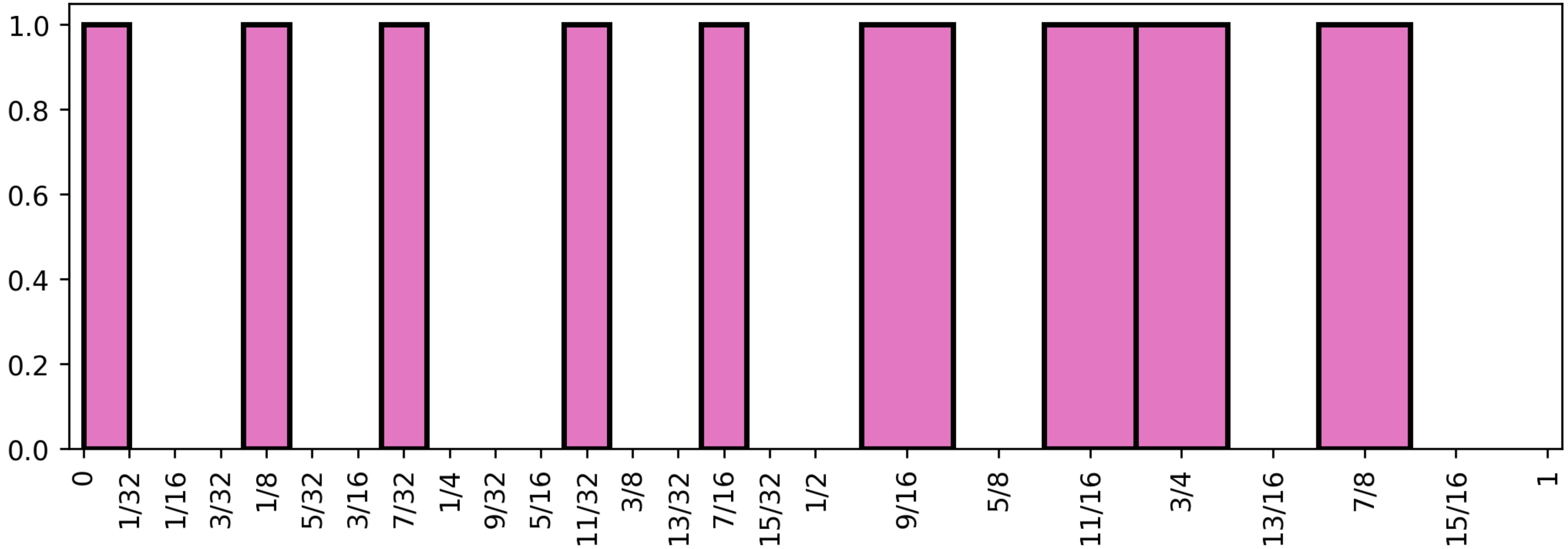
```
// GSL: Existing random variate generators for Gaussian distribution (renamed for clarity).
double gsl_ran_gaussian_inverse_cdf  (const gsl_rng *r, double sigma);
double gsl_ran_gaussian_box_muller   (const gsl_rng *r, double sigma);
double gsl_ran_gaussian_ziggurat     (const gsl_rng *r, double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

Example: A correct uniform over all floats in (0,1)

Traditional Method (Dividing Integers)

- covers <1% of floats, many gaps
- may have incorrect probabilities
- bit patterns have undesirable properties
- see also Goualard (2020)

```
double unif_gen(){
  i = rand();
  return i / (RAND_MAX+1.0);
}
```

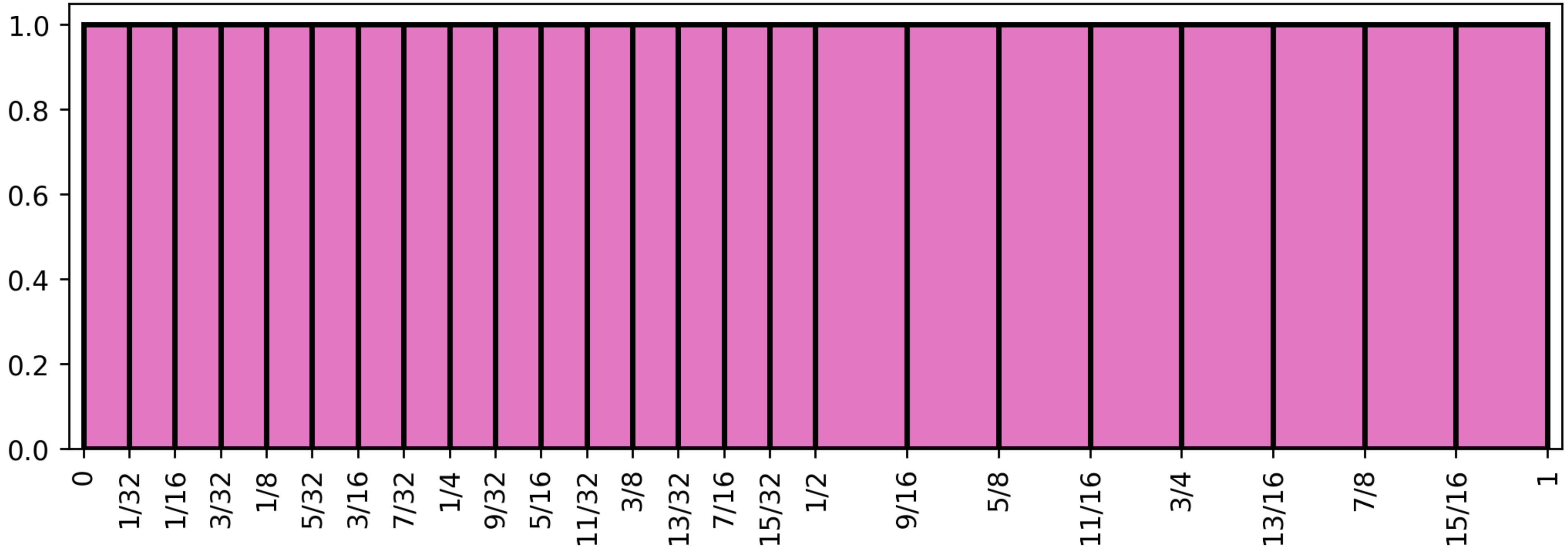


Example: A correct uniform over all floats in (0,1)

Proposed Method (Synthesize from CDF)

- covers 100% of floats (incl. subnormals)
- guarantees correctly rounded probabilities
- automated (custom solutions exist)

```
double unif_cdf(double x) {
  if      (x<0)  {return 0;}
  else if (x<=1) {return x;}
  else           {return 1;}
}

GENERATE_FROM_CDF(unif_cdf)
```



Comparison of Traditional and Proposed Approach

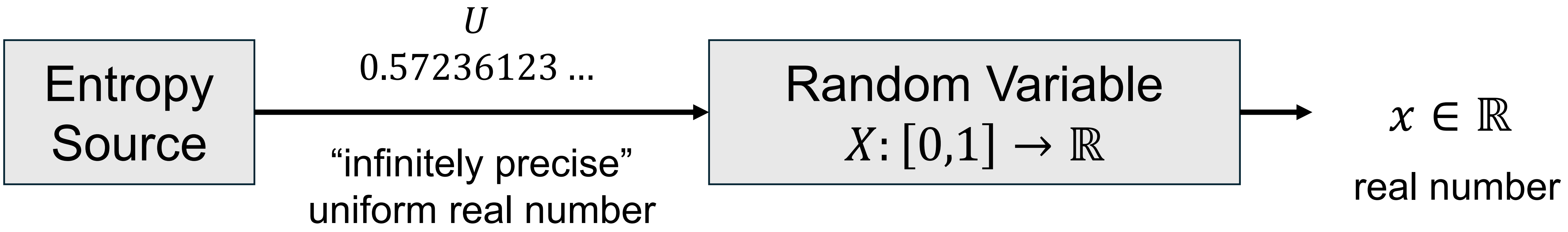
	Traditional Approach	Proposed Approach
Automation	**separate implementations** of the CDF, SF, QF, and RVG	**automatically synthesize** the RVG (and QF/SF) given CDF spec
Coherence	CDF/SF/QF/RVG all **disagree**	CDF/SF/QF/RVG all **agree**
Entropy	highly **wasteful** of random bits	information-theoretically **optimal** cost
Analyzability	output distribution of the RVG is **intractable** to compute	output distribution of the RVG **exactly matches** its formal spec
Accuracy	covers **narrower** range of values	covers **broader** range of values

Agenda

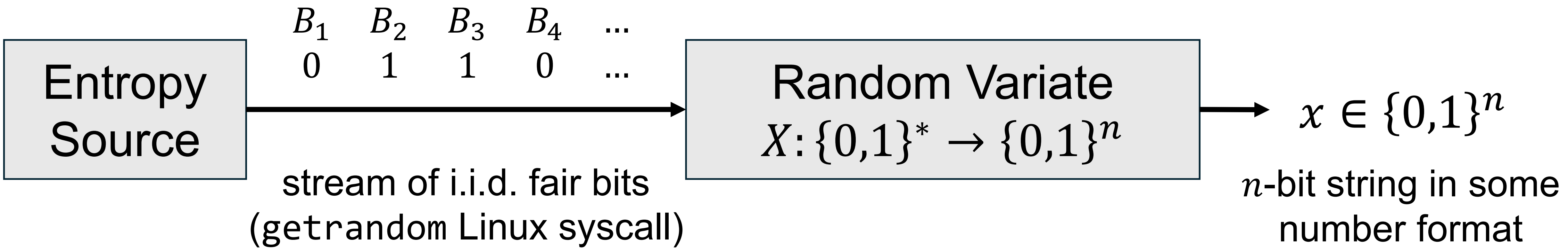
- Overview of Random Variate Generation
- **Technical Approach**
- Experimental Results
- Future Work

The random bit model of computation

Infinite-Precision Model



Finite-Precision Model



DDG trees: A universal computational model

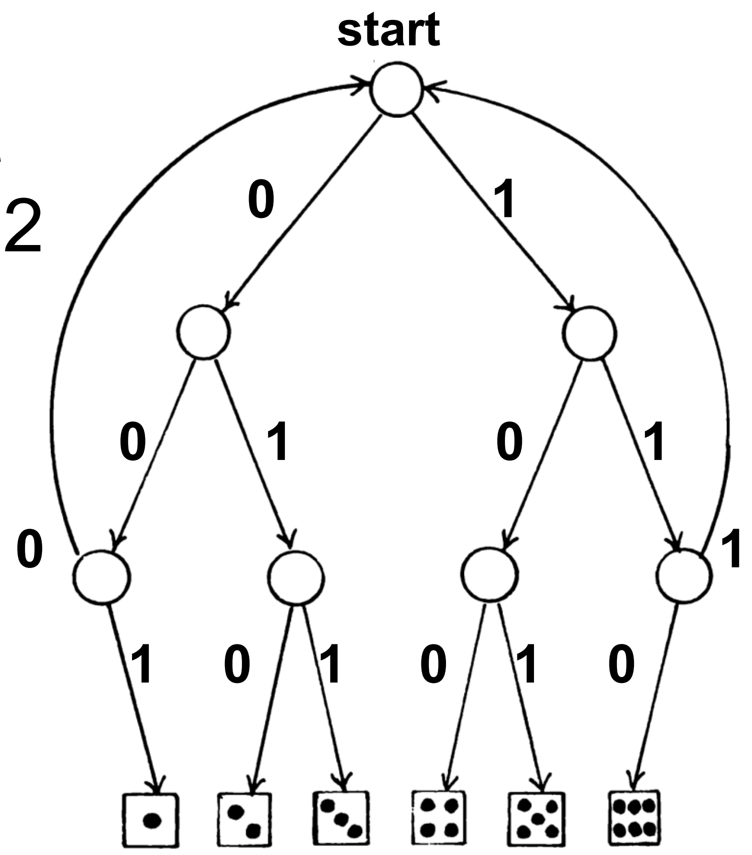
Suppose the target distribution $p = (p_1, \dots, p_n)$ is discrete

Every computable RVG is a **discrete distribution generating (DDG)** tree T:

1. start at the root node
2. get a new random bit: if 0 go left, else go right
3. if reach a leaf node, return its label, else goto 2

$$p = (1/6, \dots, 1/6)$$

$X(001) = 1$	$X(101) = 5$
$X(010) = 2$	$X(110) = 6$
$X(011) = 3$	$X(000b) = X(b)$
$X(100) = 4$	$X(111b) = X(b)$



Properties of DDG trees

- The **output distribution** of a DDG tree T is determined by leaf labels:

$$P_T(i) := \sum_{l \in \text{leaves}(T)} 2^{-\text{depth}(l)} \cdot \mathbb{I}[\text{label}(l) = i]$$

- The **entropy cost** of a DDG tree T is the average no. of consumed flips

$$C_T := \sum_{l \in \text{leaves}(T)} 2^{-\text{depth}(l)} \cdot \text{depth}(l)$$

Entropy-optimal DDG trees

Goal For a distribution $p = (p_1, \ldots, p_n)$, construct a DDG tree T^* such that

- T^* has output distribution p — $P_{T^*} \equiv p$
- T^* has minimal possible entropy cost — $C_{T^*} = \min_T \{C_T \mid P_T = p\}$

Entropy-optimal DDG trees

Goal For a distribution $p = (p_1, \dots, p_n)$, construct a DDG tree T^* such that

- T^* has output distribution p $\quad P_{T^*} \equiv p$
- T^* has minimal possible entropy cost $\quad C_{T^*} = \min_T \{C_T \mid P_T = p\}$

Theorem (Knuth & Yao 1976)

Let $p_i = (p_{i0}.p_{i1}p_{i2}p_{i3} \dots)_2$ be the binary expansions of the p_i $(i \in [n])$.

T^* has a leaf with label i at depth j **if and only if** $p_{ij} = 1 \; (\mathrm{i} \in [n], j \geq 0)$.

Gives a constructive procedure for building entropy-optimal DDG trees!

Technical challenges with entropy-optimal DDG trees

Theorem (Knuth & Yao 1976)

Let $p_i = (p_{i0}.p_{i1}p_{i2}p_{i3} \dots)_2$ be the binary expansions of the p_i $(i \in [n])$.

T^* has a leaf with label i at depth j **if and only if** $p_{ij} = 1$.

Challenge 1 Even if n is small, T can have exponential depth (Saad et al., POPL 2020)

Challenge 2 The RVG $X: \{0,1\}^* \to \{0,1\}^k$ has $n = 2^k$ outcomes, so $|T| \geq 2^k$. For distribution over 64-bit floats, $n = 2^{64}$

Cannot hope to explicitly construct the entire DDG tree

Key idea: Binary-coded probability distribution

Any distribution over $\mathbb{R}$ is a CDF $F: \mathbb{R} \to [0,1]$

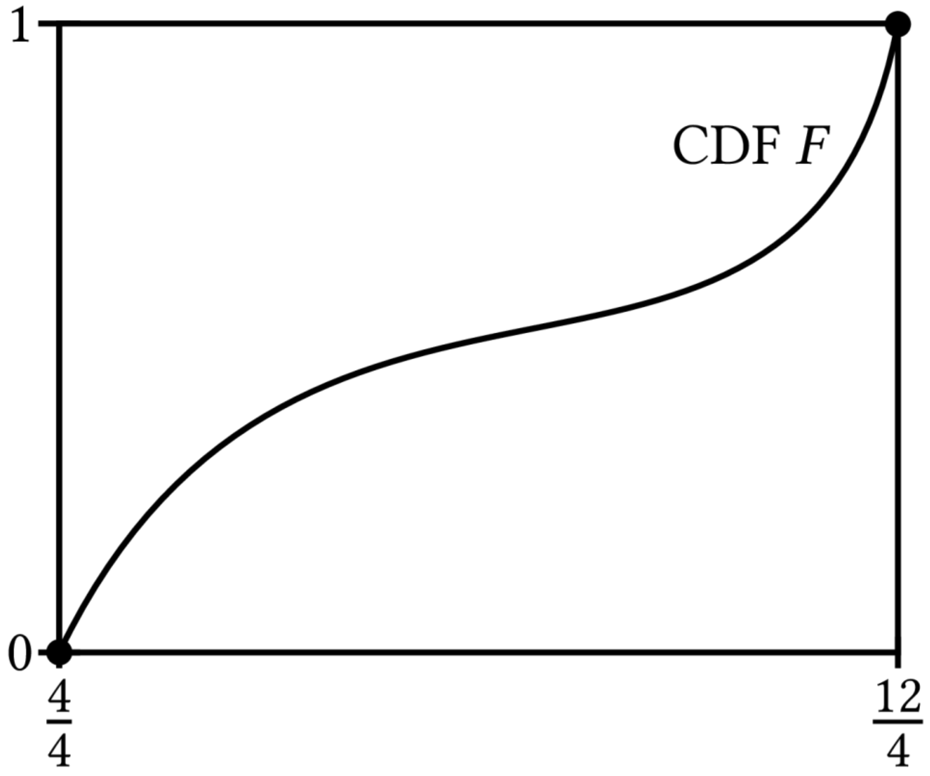
Binary-Coded Probability Distribution

alternative representation $p: \{0,1\}^* \to [0,1]$

- $p(\varepsilon) = 1$
- $p(b0) + p(b1) = p(b), \forall b \in \{0,1\}^*$

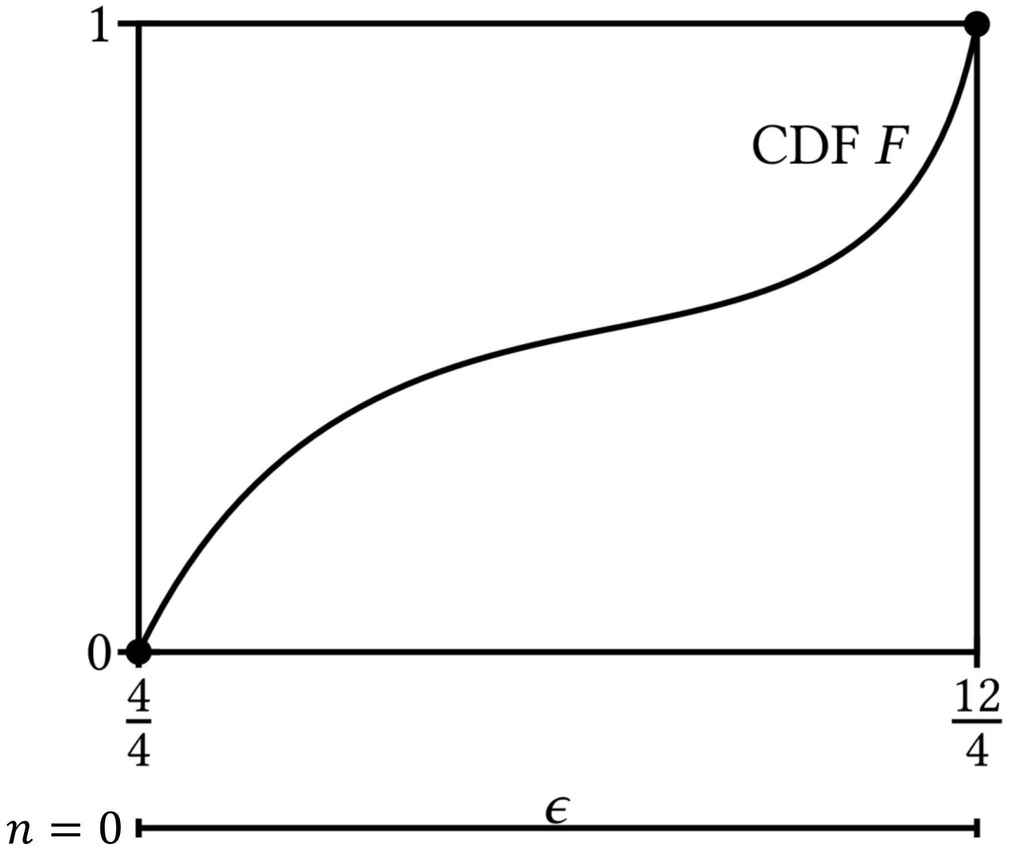
p encodes F as a sequence of discrete probability distributions over $\{0,1\}^n, n \geq 0$

$\{0,1\}^*$ encodes recursive partitions of $\mathbb{R}$



Key idea: Binary-coded probability distribution

Any distribution over $\mathbb{R}$ is a CDF $F: \mathbb{R} \to [0,1]$



Binary-Coded Probability Distribution
alternative representation $p: \{0,1\}^* \to [0,1]$

- $p(\varepsilon) = 1$
- $p(b0) + p(b1) = p(b), \forall b \in \{0,1\}^*$

p encodes F as a sequence of discrete probability distributions over $\{0,1\}^n, n \geq 0$

$\{0,1\}^*$ encodes recursive partitions of $\mathbb{R}$

Key idea: Binary-coded probability distribution

Any distribution over $\mathbb{R}$ is a CDF $F: \mathbb{R} \to [0,1]$

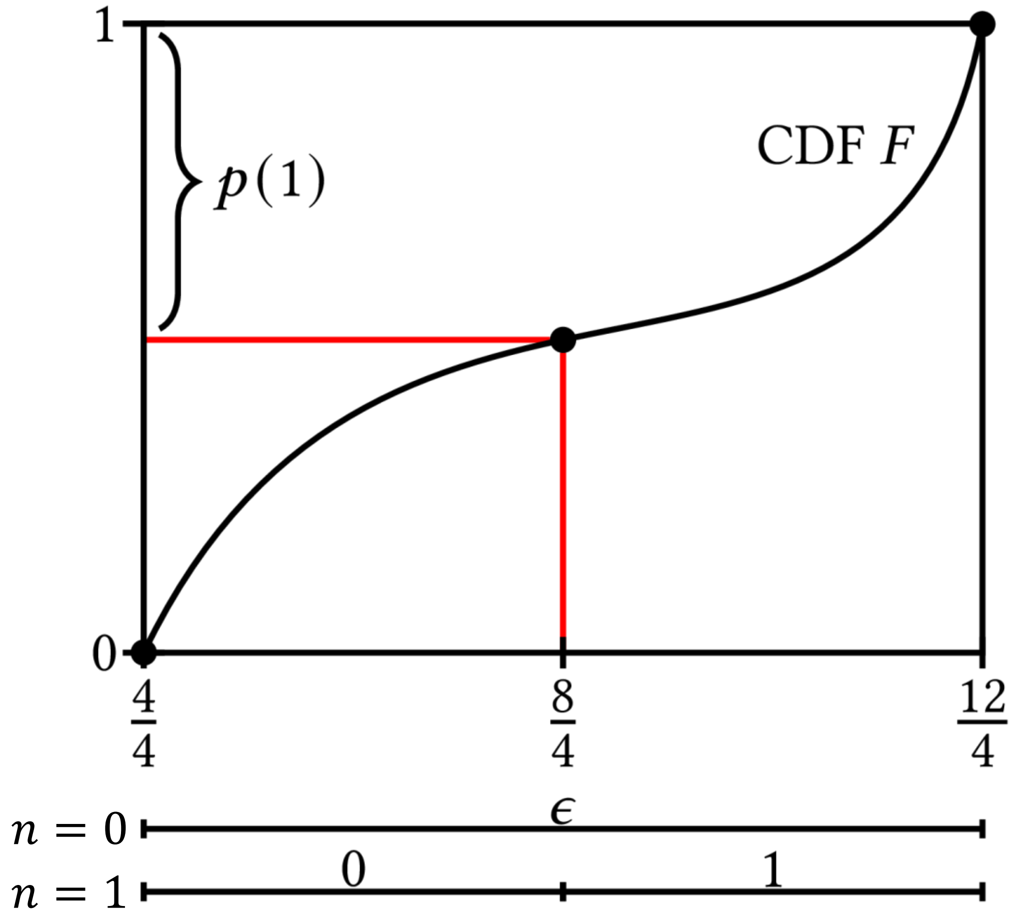
Binary-Coded Probability Distribution

alternative representation $p: \{0,1\}^* \to [0,1]$

- $p(\varepsilon) = 1$
- $p(b0) + p(b1) = p(b), \forall b \in \{0,1\}^*$

p encodes F as a sequence of discrete probability distributions over $\{0,1\}^n, n \geq 0$

$\{0,1\}^*$ encodes recursive partitions of $\mathbb{R}$



Key idea: Binary-coded probability distribution

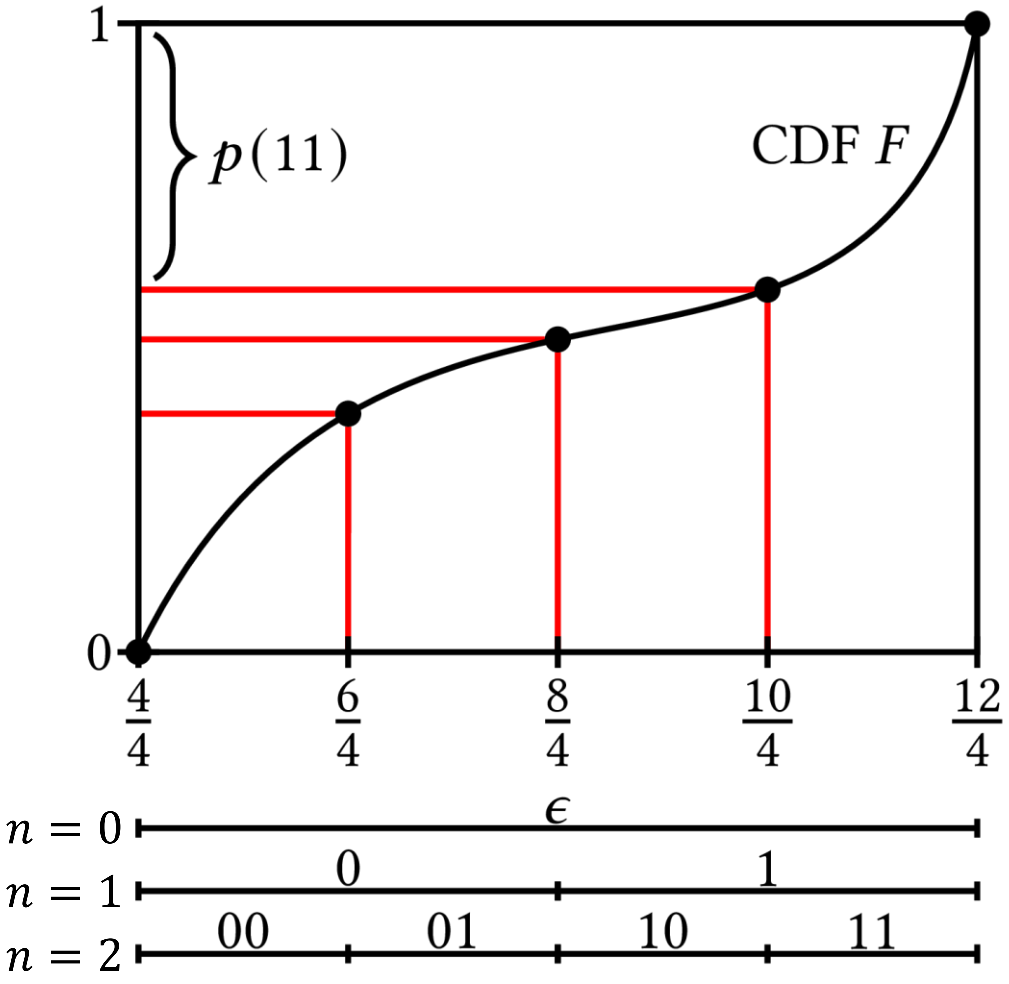
Any distribution over $\mathbb{R}$ is a CDF $F: \mathbb{R} \to [0,1]$

<u>**Binary-Coded Probability Distribution**</u>
alternative representation $p: \{0,1\}^* \to [0,1]$

- $p(\varepsilon) = 1$
- $p(b0) + p(b1) = p(b), \forall b \in \{0,1\}^*$

p encodes F as a sequence of discrete probability distributions over $\{0,1\}^n, n \geq 0$

$\{0,1\}^*$ encodes recursive partitions of $\mathbb{R}$



Key idea: Binary-coded probability distribution

Any distribution over $\mathbb{R}$ is a CDF $F: \mathbb{R} \to [0,1]$

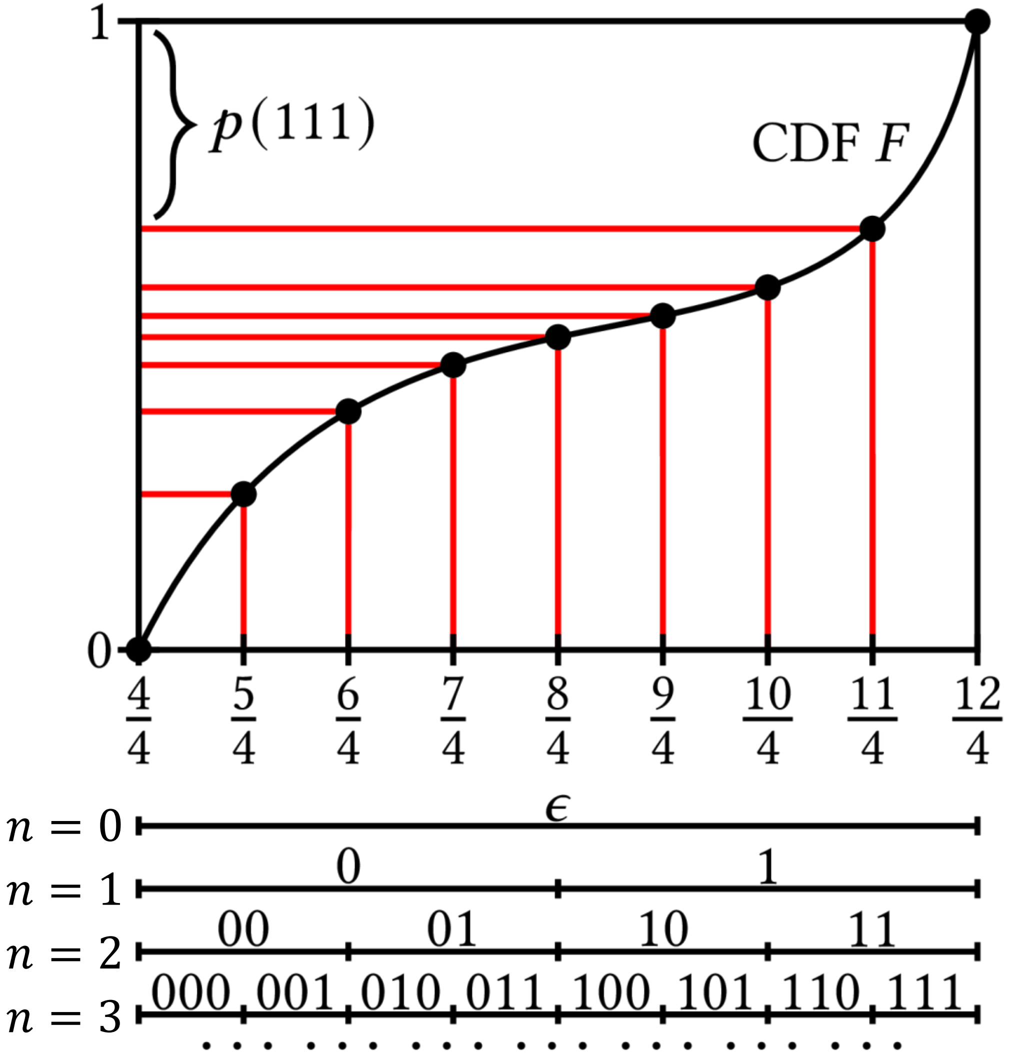
Binary-Coded Probability Distribution

alternative representation $p: \{0,1\}^* \to [0,1]$

- $p(\varepsilon) = 1$
- $p(b0) + p(b1) = p(b), \forall b \in \{0,1\}^*$

p encodes F as a sequence of discrete probability distributions over $\{0,1\}^n, n \geq 0$

$\{0,1\}^*$ encodes recursive partitions of $\mathbb{R}$

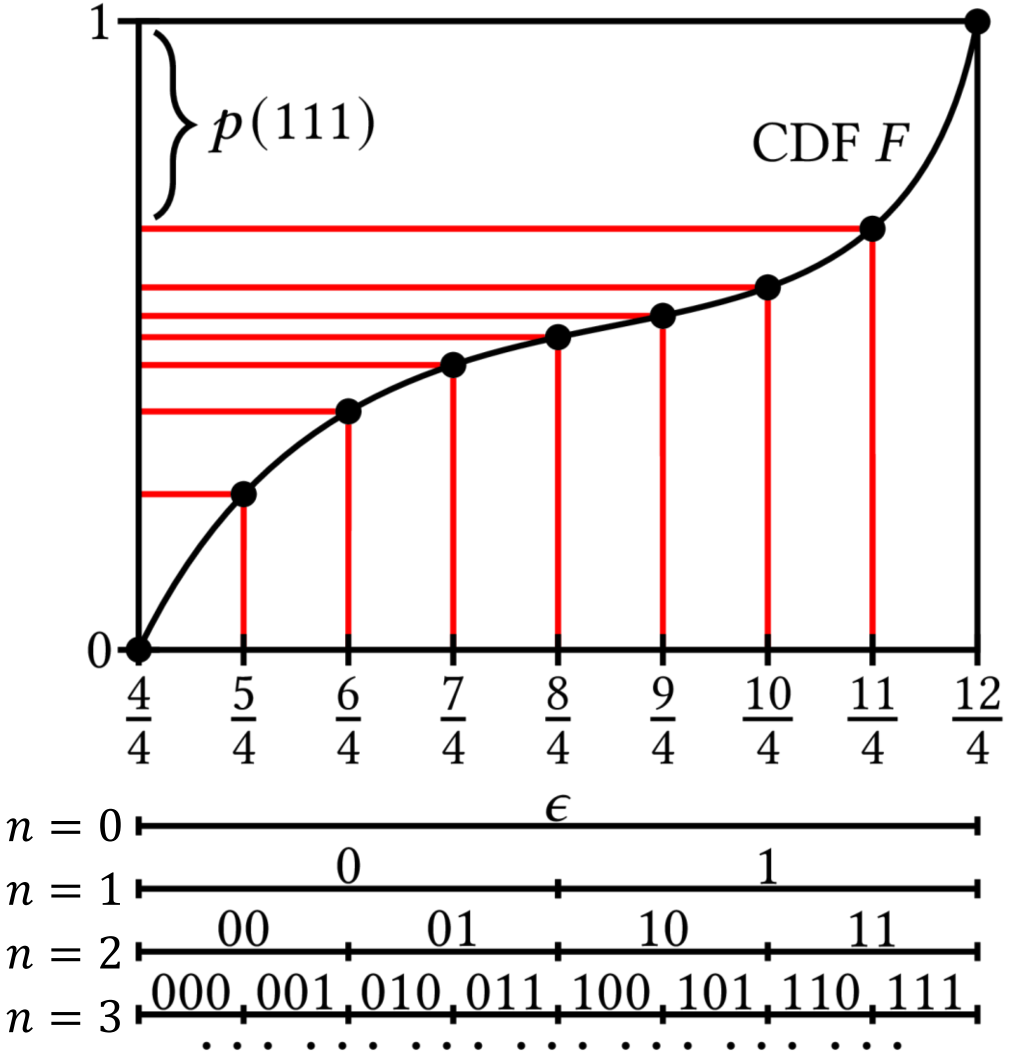


Drawing bits from a binary-coded distribution

Given a binary-coded probability distribution $p: \{0,1\}^* \to [0,1]$, generate random bits

$$B_1 B_2 B_3 \ldots B_n \sim p$$

Can map $B_1 \ldots B_n$ to a point/interval of $\mathbb{R}$



Drawing bits from a binary-coded distribution

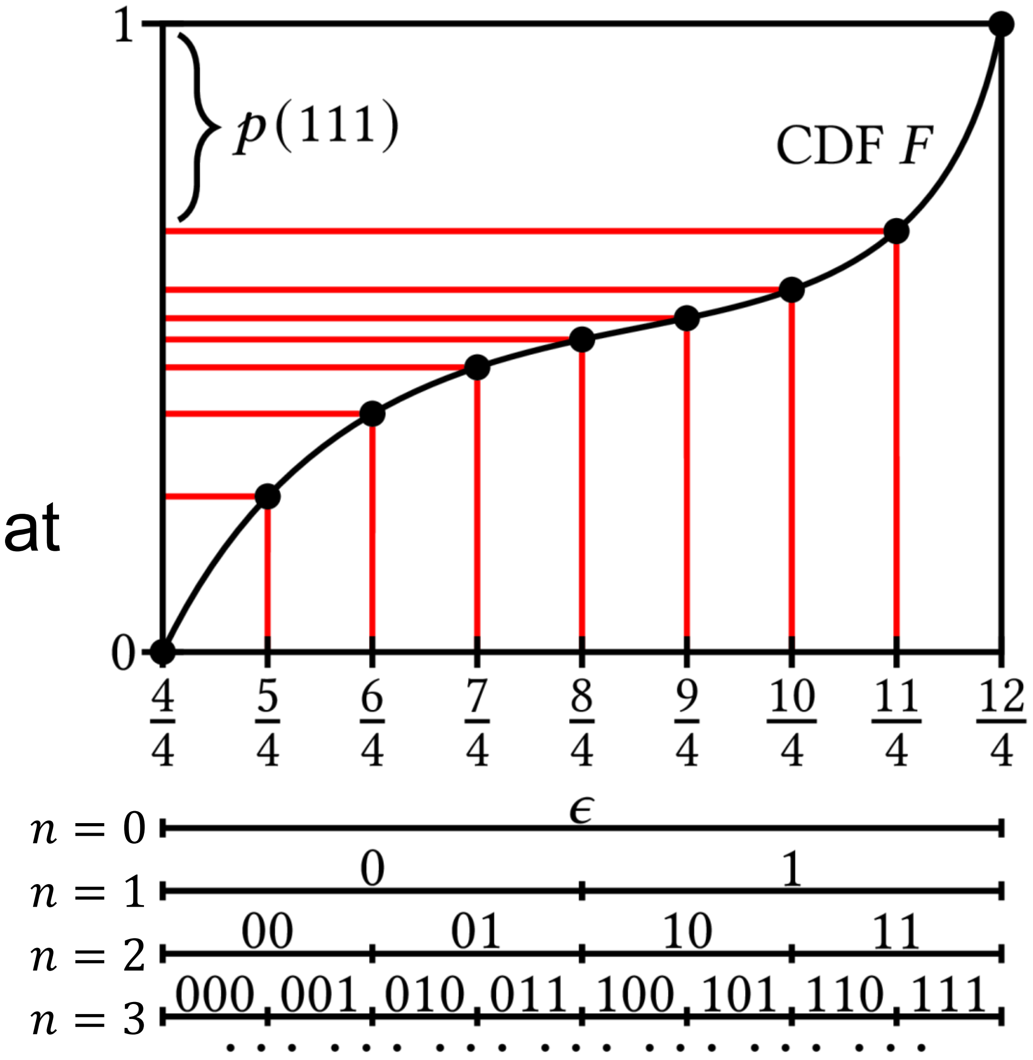
Given a binary-coded probability distribution $p: \{0,1\}^* \to [0,1]$, generate random bits

$$B_1 B_2 B_3 \ldots B_n \sim p$$

Can map $B_1 \ldots B_n$ to a point/interval of $\mathbb{R}$

Also works if $\{0,1\}^n$ is any binary-number format

Integers	Rationals
Unsigned Integer	Fixed Point
Sign and Magnitude	Floating Point
Two's Complement	Posits



Drawing bits from a binary-coded distribution

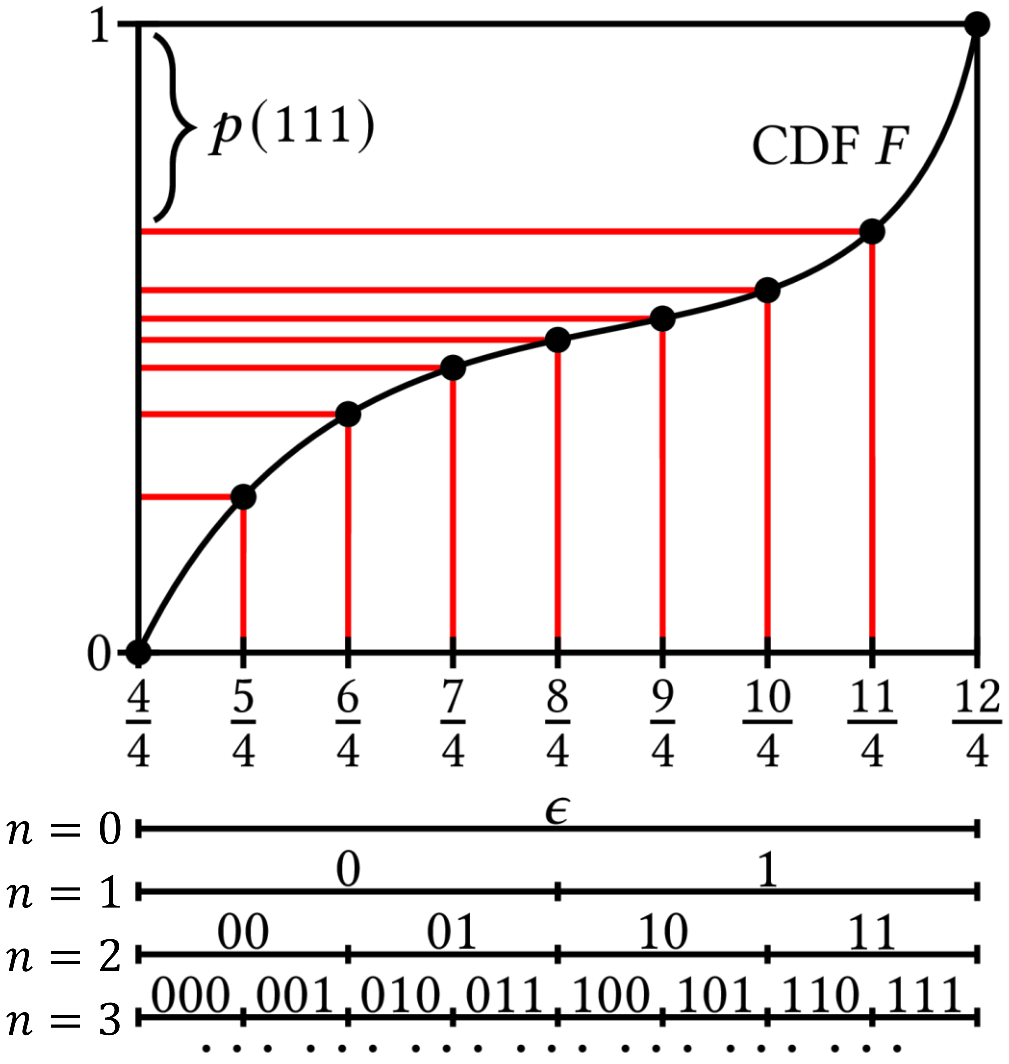
Given a binary-coded probability distribution $p: \{0,1\}^* \to [0,1]$, generate random bits

$$B_1 B_2 B_3 \dots B_n \sim p$$

<u>Naïve Baseline</u> (Conditional Bit Sampling)

- $B_1 \sim \text{Bernoulli}(p(1))$
- $B_2 | B_1 \sim \text{Bernoulli}\left(\frac{p(B_1 1)}{p(B_1)}\right)$
- $B_3 | B_1, B_2 \sim \text{Bernoulli}\left(\frac{p(B_1 B_2 1)}{p(B_1 B_2)}\right)$
- ...

Highly suboptimal in space, runtime, entropy



Contribution 1: Space-time-entropy optimal generator

Given a binary-coded probability distribution $p: \{0,1\}^* \to [0,1]$, generate random bits

$$B_1 B_2 B_3 \ldots B_n \sim p$$

<u>Optimal Method</u> (Our Method)

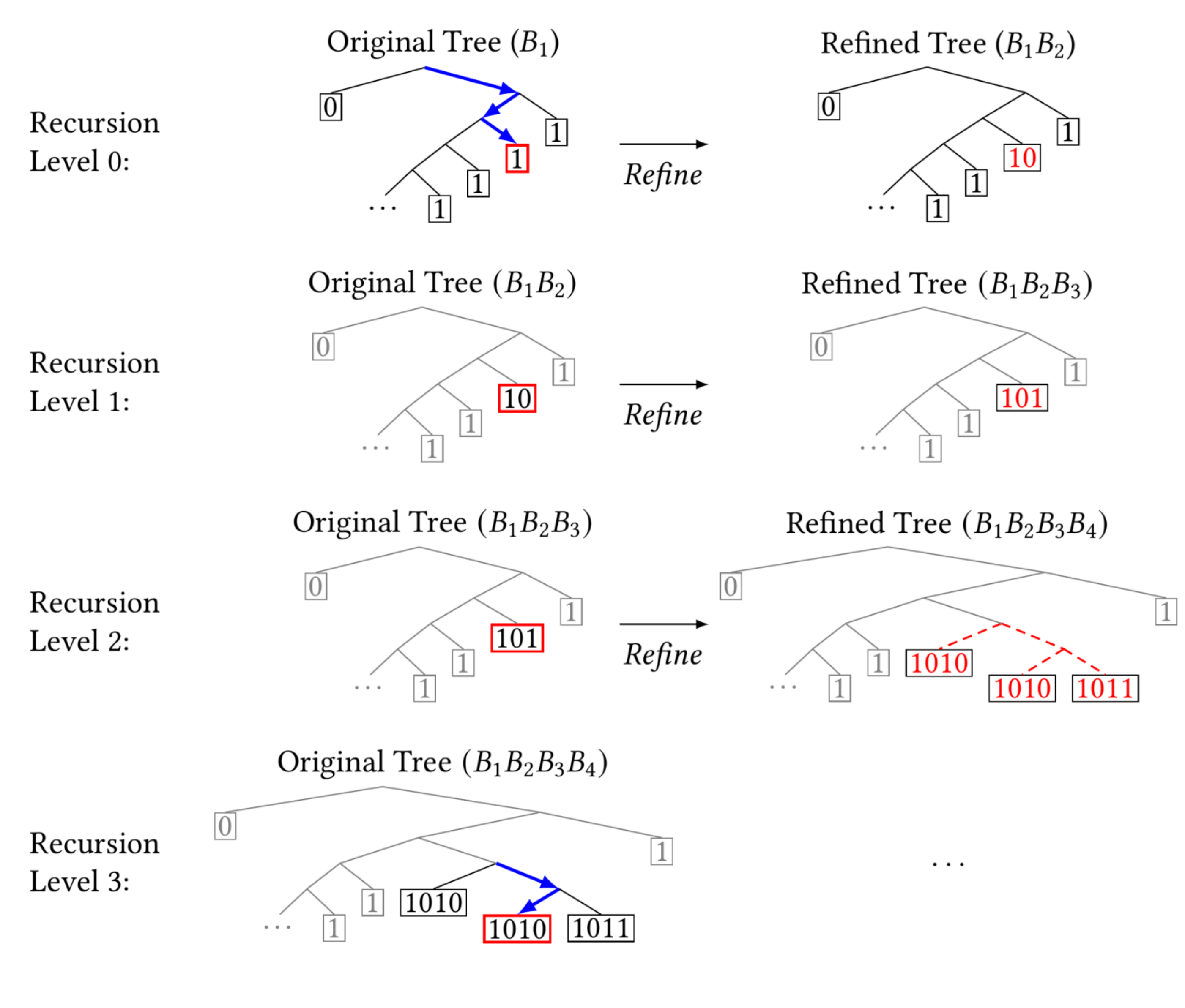
Lazily explores an entropy-optimal tree

Correctness proof leverages analytic properties of unit interval $[0,1] \subset \mathbb{R}$

Optimal in *space*, *time*, and *entropy*

```
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'))
```

Contribution 1: Space-time-entropy optimal generator

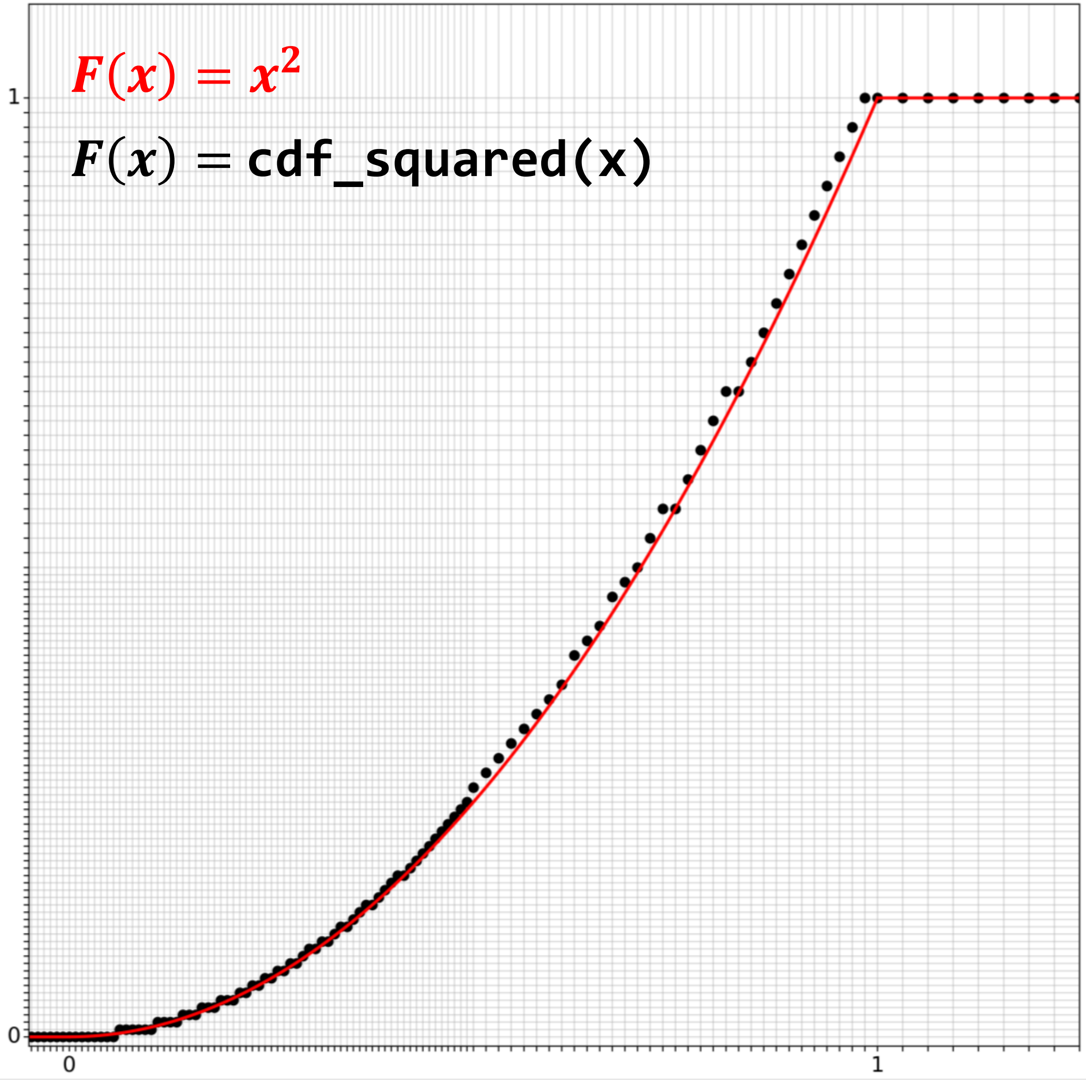



```haskell
type Bit = Int
type BinaryString = [Bit]
type BinaryCodedDist = [Bit] -> Float

-- 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'))
```

Contribution 2: Efficient floating-point implementation

But a numerical CDF $F: \{0,1\}^n \to \mathbb{F} \cap [0,1]$ has floating-point probabilities!

```
double cdf_squared(double x) {
 if      (x<0)  {return 0;}
 else if (x<=1) {return x*x;}
 else           {return 1;}
}
```



Contribution 2: Efficient floating-point implementation

But a numerical CDF $F: \{0,1\}^n \to \mathbb{F} \cap [0,1]$ has floating-point probabilities!

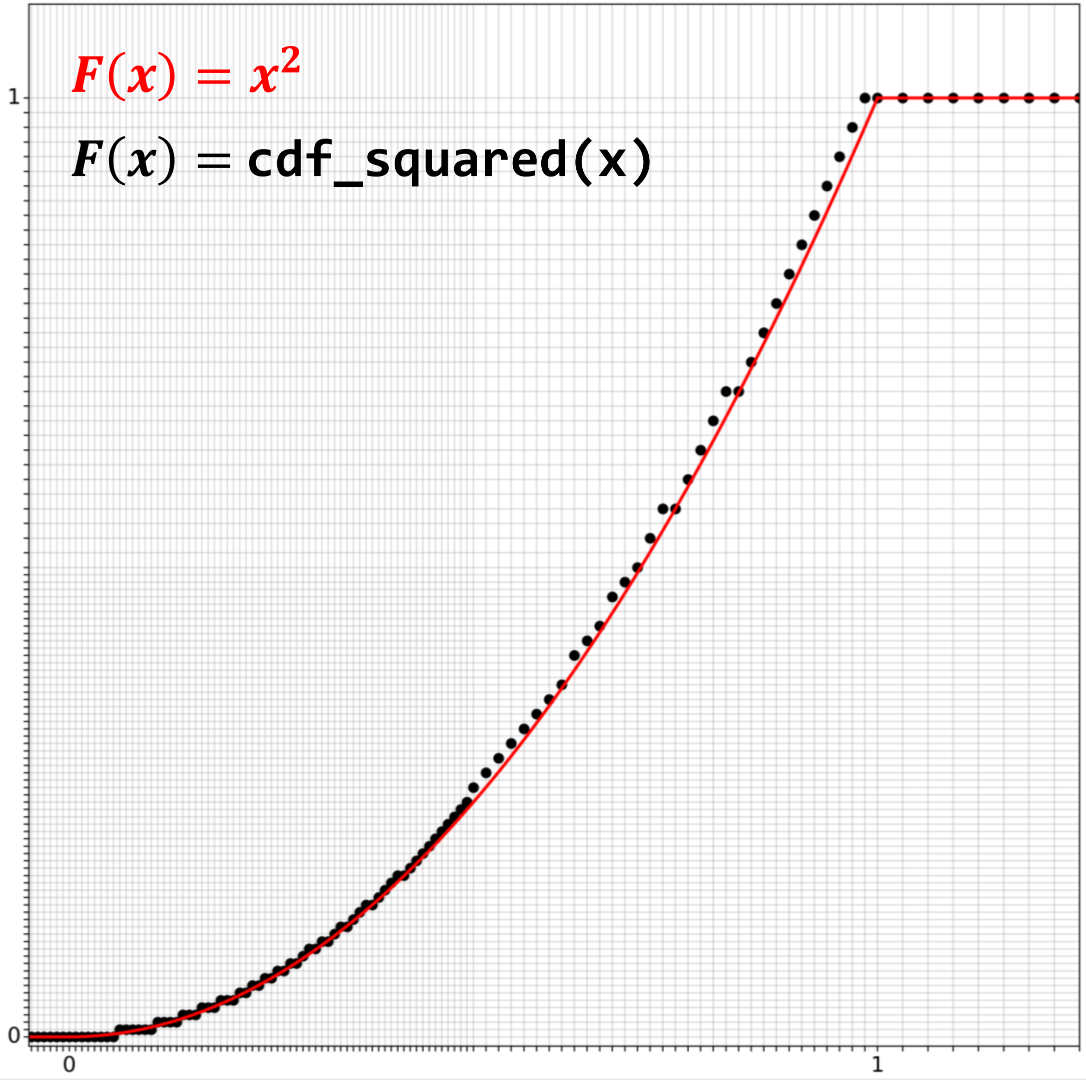
```
double cdf_squared(double x) {
 if       (x<0)  {return 0;}
 else if (x<=1) {return x*x;}
 else            {return 1;}
}
```

We give an exact floating-point generator for $p: \{0,1\}^* \to \mathbb{F}$

- ✓ uses fast integer arithmetic
- ✓ uses same precision level as F

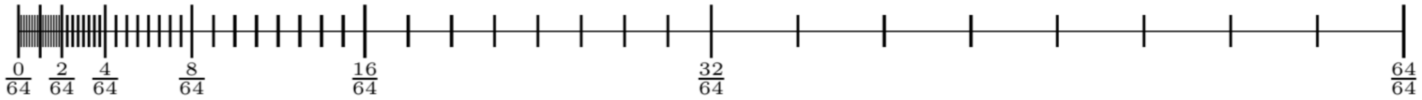
Technical challenge

- $f_1, f_2 \in \mathbb{F} \not\Rightarrow f_2 -_{\mathbb{R}} f_1 \in \mathbb{F}$
- Need binary expansion of $f_2 -_{\mathbb{R}} f_1$

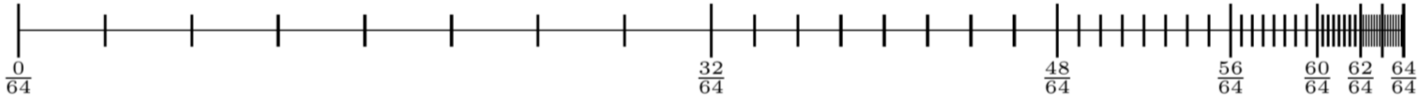


Contribution 3: Extended accuracy with survival functions

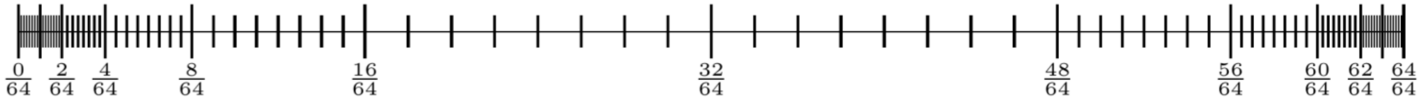
Cumulative probabilities in $\mathbb{F}$
$F(x)$ are inaccurate near 1



Tail probabilities in $\mathbb{F}$: use a
survival function $S(x) = 1 - F(x)$



Combine the two functions into a
dual distribution function

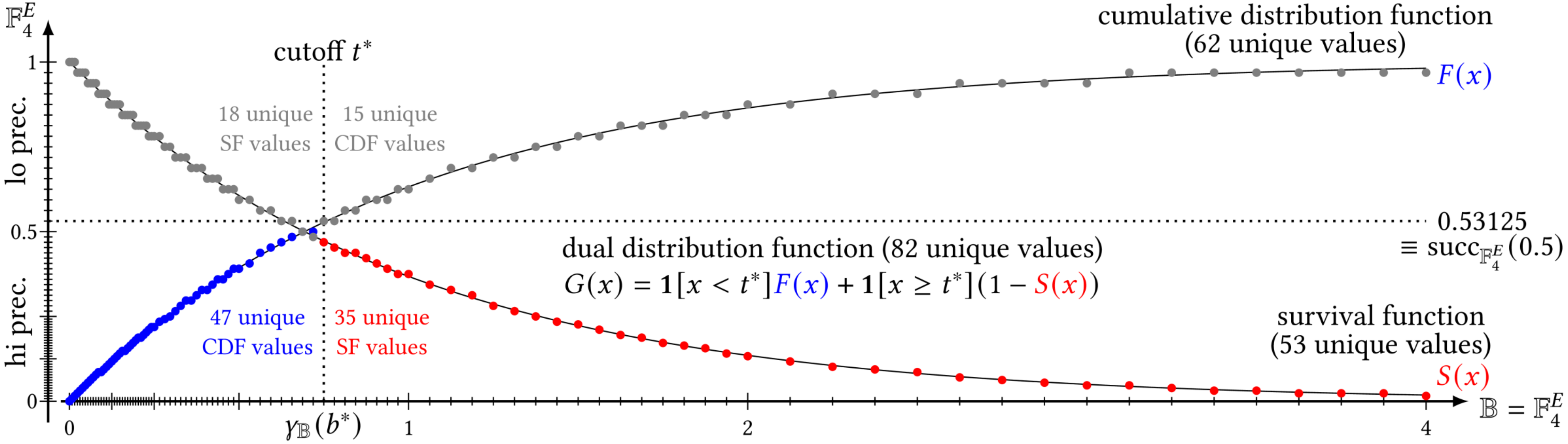



```
standard_rayleigh_cdf = lambda t: -math.expm1(-t*t/2)
standard_rayleigh_sf  = lambda t: math.exp(-t*t/2)
```

$[3.5 \times 10^{-162}, 8.65]$
$[1.05 \times 10^{-8}, 38.6]$

<u>Combined Representation</u> $[3.5 \times 10^{-162}, 38.6]$

Contribution 3: Extended accuracy with survival functions



Can represent twice as many values without increasing precision

We give an entropy-optimal generator for any DDF (see paper for details)

Agenda

- Overview of Random Variate Generation
- Technical Approach
- **Experimental Results**
- Future Work

Research Questions

- **Question 1** How do bits/variate and variates/sec compare to GSL?
- **Question 2** How does accuracy compare to GSL?
- **Question 3** What is the overhead of extended accuracy generation?

Q1: Performance in bits/variates and variates/sec

Distribution	Method	Bits/Variate
Beta(5, 5)	GSL	262.80
	CBS	52.10
	OPT	24.98
Binomial(.2, 100)†	GSL	224.79
	CBS	15.76
	OPT	5.11
Cauchy(7)	GSL	64.00
	CBS	51.45
	OPT	25.00
ChiSquare(13)	GSL	64.00
	CBS	47.43
	OPT	24.99
Exponential(15)	GSL	64.00
	CBS	48.56
	OPT	24.98
ExpPow(1, .5)	GSL	197.03
	CBS	47.31
	OPT	25.01
Fdist(5, 2)	GSL	268.95
	CBS	51.45
	OPT	25.00
Flat(-7, 3)	GSL	64.00
	CBS	43.52
	OPT	24.98
Gamma(.5, 1)	GSL	198.26
	CBS	57.00
	OPT	25.01
Gaussian(15)	GSL	162.73
	CBS	46.41
	OPT	25.00
Geometric(.4)†	GSL	64.00
	CBS	6.06
	OPT	3.78
Gumbel1(1,1)	GSL	64.00
	CBS	50.29
	OPT	25.00

Distribution	Method	Bits/Variate
Gumbel2(1, 5)	GSL	64.00
	CBS	49.26
	OPT	24.99
Hypergeom(5, 20, 7)†	GSL	447.99
	CBS	6.25
	OPT	3.01
Laplace(2)	GSL	64.00
	CBS	47.83
	OPT	25.00
Logistic(.5)	GSL	64.00
	CBS	48.80
	OPT	24.97
Lognormal(1, 1)	GSL	163.02
	CBS	49.27
	OPT	24.98
NegBinomial(.71, 18)†	GSL	665.83
	CBS	12.54
	OPT	4.69
Pareto(3,2)	GSL	64.00
	CBS	45.92
	OPT	24.99
Pascal(1, 5)†	GSL	195.59
	CBS	0.00
	OPT	0.00
Poisson(71)†	GSL	697.22
	CBS	18.32
	OPT	6.19
Rayleigh(11)	GSL	64.00
	CBS	48.52
	OPT	24.99
Tdist(5)	GSL	279.77
	CBS	49.56
	OPT	25.02
Weibull(2, 3)	GSL	64.00
	CBS	55.35
	OPT	24.97

GSL = GNU Scientific Library
CBS = Naïve Baseline Generator
OPT = Optimal Generator

bits/variate (lower = better)

OPT 1x–3x better than CBS
OPT 3x–142x better than GSL

Q1: Performance in bits/variates and variates/sec

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

GSL = GNU Scientific Library
CBS = Naïve Baseline Generator
OPT = Optimal Generator

bits/variate (lower = better)

OPT 1x–3x better than CBS
OPT 3x–142x better than GSL

variates/sec (higher = better)

OPT 1x–9x faster than CBS
OPT 2x–35x slower than GSL (median 6x)

Q2: Accuracy of generated variates

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

GSL GNU Scientific Library
CDF Cumulative Distribution Function
SF Survival Function
DDF Dual Distribution Function

Key Takeaways

DDF vs GSL — up to 10^{35}x wider coverage of range

DDF vs CDF/SF — fixes asymmetries in the tail accuracy

Q2: Accuracy of generated variates

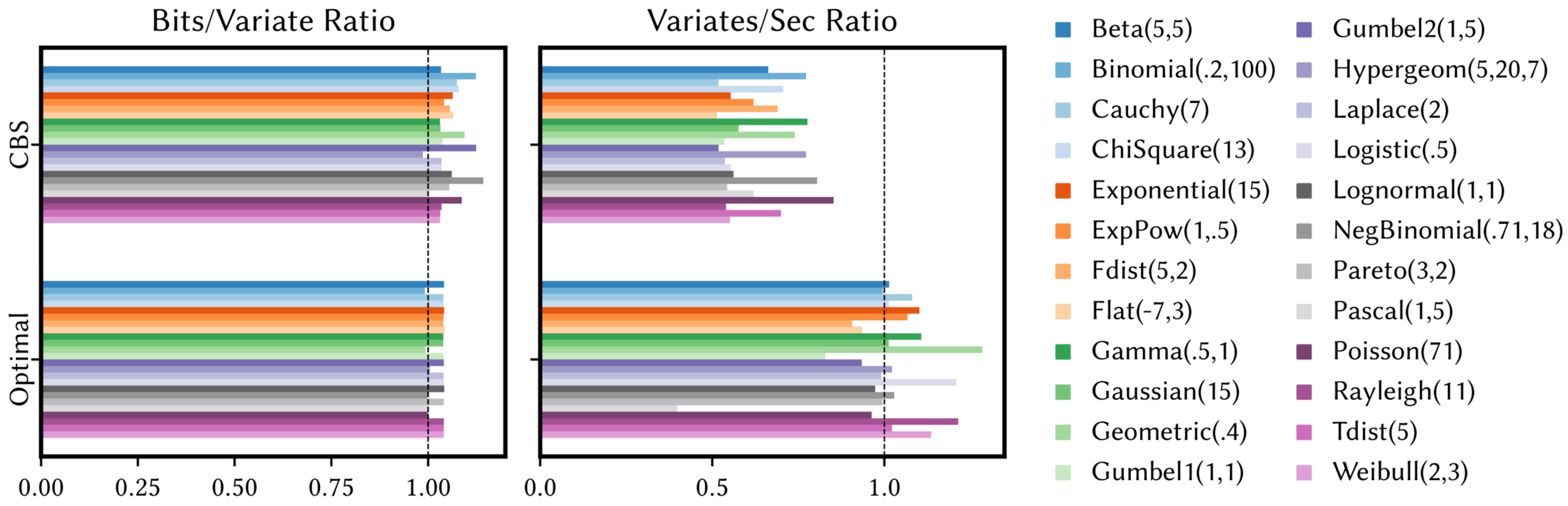
Distribution	Method	Random Variate Range			Analysis Time
Gamma(.5, 1) $(0, \infty)$	GSL	—	unknown	—	∞
	CDF	3.86×10^{-91}		15.36	<50 µs
	SF	6.98×10^{-16}		101.09	<50 µs
	DDF	3.86×10^{-91}		101.09	<50 µs
Gaussian(0, 1) $(-\infty, \infty)$	GSL	—	unknown	—	∞
	CDF	-14.17		5.42	<50 µs
	SF	-5.42		14.17	<50 µs
	DDF	-14.17		14.17	<50 µs
Tdist(1) $(-\infty, \infty)$	GSL	—	unknown	—	∞
	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

GSL — GNU Scientific Library
CDF — Cumulative Distribution Function
SF — Survival Function
DDF — Dual Distribution Function

Key Takeaways

DDF vs GSL — GSL is often intractable to analyze

Q3: Overhead of extended accuracy generation



Minimal overhead for OPT, high overhead for CBS (Naïve Baseline)

Agenda

- Overview of Random Variate Generation
- Technical Approach
- Experimental Results
- **Future Work**

Future Work

formally verify that a numerical CDF $F: \{0,1\}^n \to \mathbb{F}$ is valid

integrate as primitives an end-to-end verified PPL

reduce performance gap with (ad-hoc) GSL generators

parallel / vectorized implementation

quantify theoretical error between numerical and analytic CDF

applications in differential privacy, cryptography, etc.

Main Contributions

- **Precise formulation** of synthesizing exact generators given a numerical specification
- **Space-time-entropy** optimal generation for arbitrary distributions over $\mathbb{R}$
- **Exact implementation** in any finite-precision number format (integer, fixed-point, float, posit)
- **Extended-accuracy** generators that coherently combine a numerical CDF and SF
- **Improvements** over GNU Scientific Library generators https://github.com/probsys/librvg