Universal Coding of the Reals using Bisection

Conference for Next Generation Arithmetic 2019

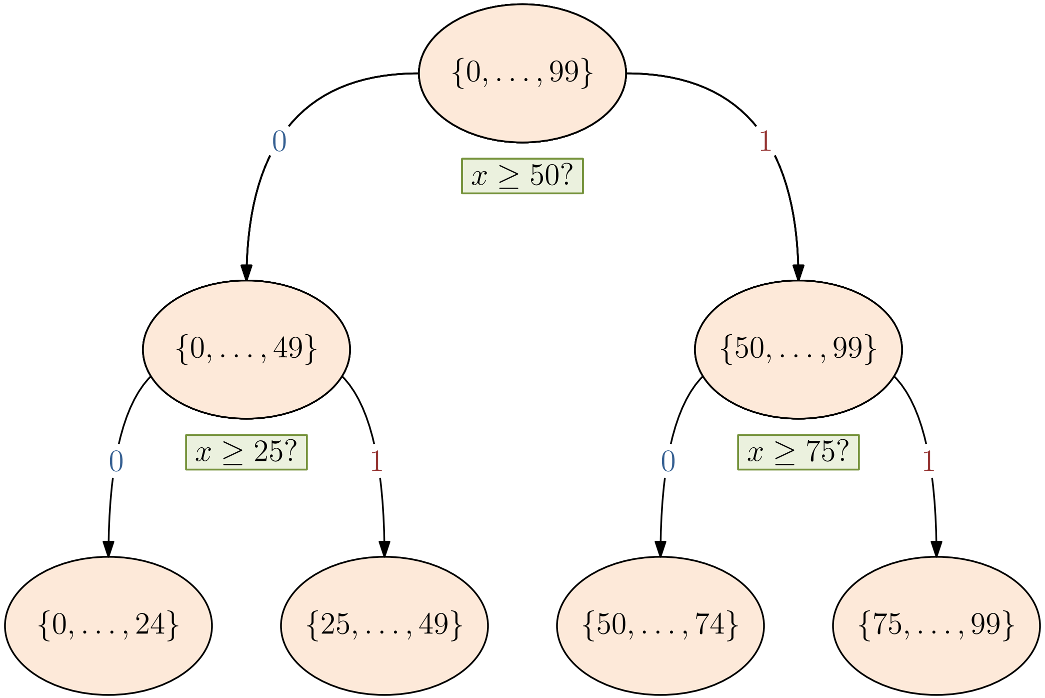
Peter Lindstrom

March 13, 2019

LLNL-PRES-769458

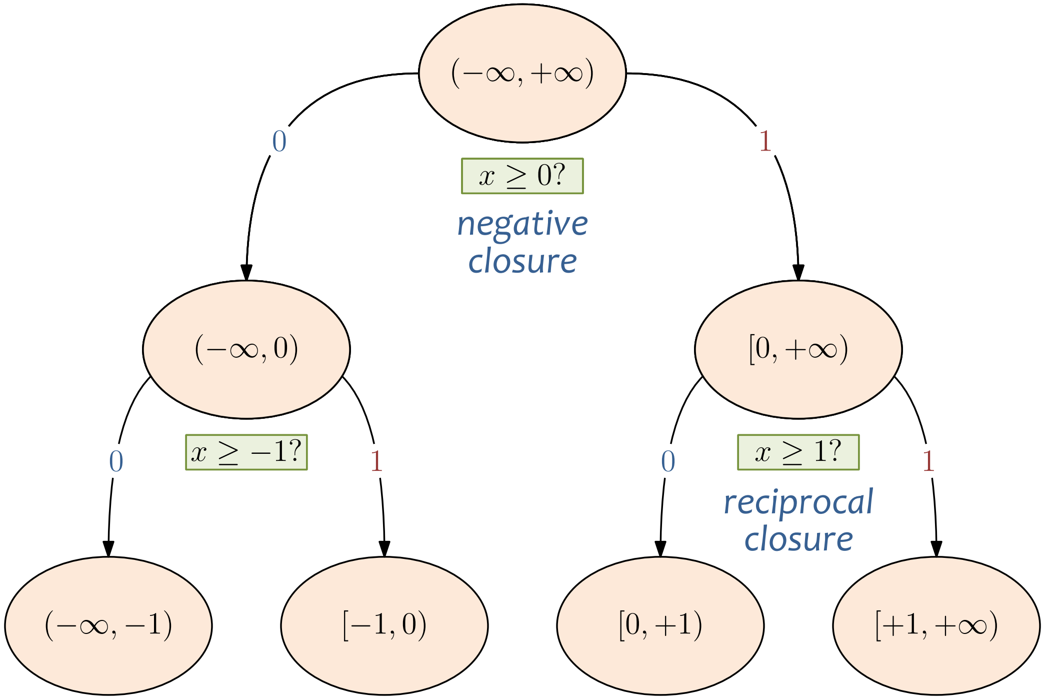
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

I’m thinking of an integer $0 \le x \le 99$. Can you guess it?



Binary search to the rescue—the outcome of each comparison corresponds to a bit in the binary representation of x

I'm thinking of a real number $-\infty < x < +\infty$. Can you guess it?



Now what?

How can we search unbounded intervals?

- Suppose we know $x \in (1, \infty)$
- Q: What should our next guess be?
- A: *Unbounded search* [Bentley & Yao 1976]
  - Step 1: Bracket x via monotonic sequence $\{a_i\}$ of guesses, e.g., $(1, 2, 4, 8, 16, \ldots)$
  - Step 2: Once $x \in [a_i, a_{i+1})$, proceed with binary search
  - Each $x \geq a_i$ comparison encodes one bit of information
- *Any* real number may be encoded this way (given enough bits of precision)
  - **Reciprocal sequence** $\{a_{-i}\} = \{1 / a_i\}$ is used when $x \in (0, 1)$, e.g., $(1, 1/2, 1/4, 1/8, 1/16, \ldots)$
  - **Negative sequence** $\{-a_i\}$ is used when $x \in (-\infty, 0)$, e.g., $(-1, -2, -4, -8, -16, \ldots)$
- What's a good bracketing sequence?

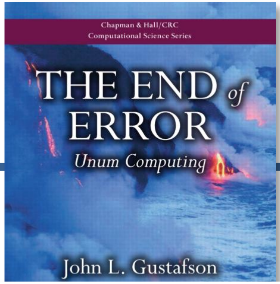
What's NOT a good bracketing sequence?

- IEEE-754 floating-point makes some curious guesses
  - $x \geq 0$ (sign bit)
  - $x \geq 2$ (roughly the exponent sign bit)
  - $x \geq 2^{513} =$ 26 815 615 859 885 194 199 148 049 996 411 692 254 958 731 641 184 786 755 447 122 887 443 528 060 147 093 953 603 748 596 333 806 855 380 063 716 372 972 101 707 507 765 623 893 139 892 867 298 012 168 192

- If x measures distance in Planck lengths, then the "best" guess after $x \geq 2$ is a googol (10^{100}) times the diameter of the universe?!

- Corrollary: IEEE greatly overestimates and has to backtrack via binary search
  - E.g., $\{a_i\} = (0, 2, 2^{513}, 2^{257}, 2^{129}, 2^{65}, 2^{33}, 2^{17}, 512, 32, 8, 4)$ when $x = 3$

POSITS [Gustafson 2017] are a new float representation that challenges IEEE



	IEEE 754 floating point	POSITS: UNUM version 3.0
Bit Length	{16, 32, 64, 128}	fixed length but arbitrary
Sign	sign-magnitude (−0 ≠ +0)	two's complement (−0 = +0)
Exponent	fixed length (biased binary)	variable length (Golomb-Rice)
Fraction Map	linear ($\varphi(x) = 1 + x$)	linear ($\varphi(x) = 1 + x$)
Infinities	{−∞, +∞}	±∞ (single point at infinity)
NaNs	many (9 quadrillion)	one
Underflow	gradual (subnormals)	gradual (natural)
Overflow	1 / FLT_TRUE_MIN = ∞ (oops!)	never (exception: 1 / 0 = ±∞)
Base	{2, 10}	$2^{2^m} \in \{2, 4, 16, 256, \dots\}$

In terms of encoding scheme, IEEE and POSITS differ in one important way

- IEEE and POSITS both represent real value as $x = (-1)^s 2^e (1 + f)$
  - s = sign, e = exponent, f = fraction (significand)

- **IEEE:** fixed-length **binary** encoding of exponent

- **POSIT**(m): variable-length **Golomb-Rice** encoding of exponent
  - $\lfloor e/2^m \rfloor + 1$ leading bits in **unary**, m trailing bits in **binary** $(e \bmod 2^m)$

Exponent	IEEE double	POSIT(3)
0	01111111111	0 000
13	10000001100	10 101
78	10001001101	1111111110 110

POSITS support more fraction bits when exponent is near zero (aka. "tapered accuracy")

Related work: Universal coding of positive integers [Elias 1975]

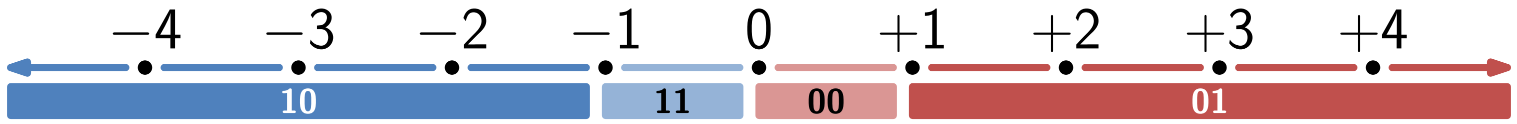
- Decompose integer $x \geq 1$ as $x = 2^e + r$
  - $e \geq 0$ is **exponent**
  - $r = x \bmod 2^e$ is e-bit **residual**

- Elias proposed three "prefix-free" variable-length codes for integers
  - Elias γ code is equivalent to POSIT(0) [Gustafson & Yonemoto 2017]
  - Elias δ code is essentially equivalent to URR [Hamada 1983]

	Elias γ code	Elias δ code	Elias ω code
Exponent	unary	γ	ω (recursive)
Code(1)	0	0	0
Code($2^e + r$)	1^e 0 r	1 $\gamma(e)$ r	1 $\omega(e)$ r
Code($17 = 2^4 + 1$)	1111 0 0001	1 11 0 00 0001	1 1 1 0 0 00 0001

At CoNGA 2018, we extended ELIAS codes from positive integers to reals

- From $z \in \mathbb{N}$ (positive integers) to $x \in \mathbb{R}$ (reals)
  - **Rationals**: $1 \leq z < x < z + 1$ encoded by appending fraction bits (as in IEEE, POSITS)
  - **Subunitaries**: $0 < x < 1$ encoded via two's complement exponent (as in POSITS)
  - **Negatives**: $x < 0$ encoded via two's complement binary representation (as in POSITS)
  - 2-bit prefix tells where we are on the real line (as in POSITS)



Beyond POSITS and Elias: NUMREP modular, templated C++ framework for generating number systems

exponent coding scheme
unary, binary, Golomb-Rice, gamma, omega, …

fraction map
linear, reciprocal, exponential, rational, …

rounding rule
round to nearest even, toward {−∞, 0, +∞}, …

overflow & underflow
snap to FLT_MAX (FLT_MIN), to infinity (zero), throw exception, …

Lindstrom, Lloyd, Hittinger, "Universal coding of the reals: Alternatives to IEEE floating point," *Conference for Next Generation Arithmetic 2018*

NUMREP unifies IEEE, POSITS, ELIAS, URR, LNS, …, under a single schema

NUMREP implementation can be tricky and error prone: Elias ω exponent coding alone is 144 lines of C++ code

```
// Elias omega exponent with at most mmax iterated exponentials
template <int mmax>
class ExpOmega : public ExpBase {
public:
  template <typename UInt>
  static ExpStatus encode(int e, UInt&
  {
    const int width = CHAR_BIT * sizeo
    ExpStatus status = expOK;
    if (e < 0) {
      // use encoding of inverted expon
      status = ExpStatus(-encode<UInt>(
      // invert lowest n bits
      x ^= UInt(~UInt(0)) >> (width - n
      // correct exponent if it does no
      if (status != expOK)
        x++;
      return status;
    }
    if (e == 0) {
      // e = 0 is encoded as binary 10
      x = 0x2u;
      n = 2;
    }
    else {
      // e = r(m) + 2^(r(m-1) + 2^(...
      // is encoded as binary 11 1^m (
      x = 0;
      n = 0;
      int m;
      for (m = 0; m < mmax && e > 1; m+
        // prepend least k = lg(e) >= 1
        int k = ilog(e);
        int l = width - k;
        if (l > 0) {
          if (((x >> k) << k) != x)
```

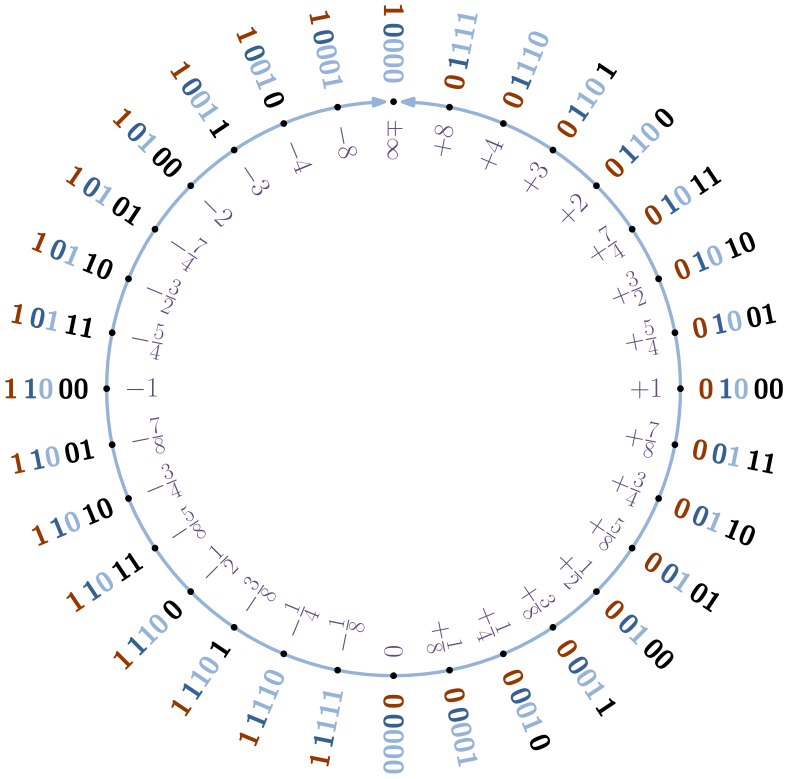
```
              status = expPosInexact;
            x >>= k;
            x += UInt(e) << l;
          }
          else {
            l = -l;
            if (x) {
              status = expPosInexact;
              x = 0;
            }
            if (((UInt(e) >> l) << l) !=
              status = expPosInexact;
            x += UInt(e) >> l;
          }
          n += k;
          e = k;
        }
        // overflow if there are unproce
        if (e > 1) {
          x = UInt(~UInt(0)) >> 1;
          n = width - 1;
          return expPosOverflow;
        }
        // prepend terminating zero-bit i
        if (m < mmax) {
          x >>= 1;
          n++;
        }
        // prepend 2 + m one-bits; length
        m += 2;
        if (((x >> m) << m) != x)
          status = expPosInexact;
        x = ~x;
        x >>= m;
        x = ~x;
        n += m;
```

```
        if (n >= width) {
          if (status == expOK)
            status = expPosTruncated;
          // round based on truncated LSB
          if (x & UInt(1)) {
            x++;
            status = expPosInexact;
            if (!x) {
              x--;
              status = expPosOverflow;
            }
          }
          x >>= 1;
          n = width - 1;
          return status;
        }
      // zero all but lowest n bits
      x >>= width - n;
    }
    return status;
  }

  template <typename UInt>
  static int decode(UInt x, int& n)
  {
    const int width = CHAR_BIT * sizeo
    int e;
    if (!msb(x)) {
      e = -decode<UInt>(~x, n) - 1;
      if (n >= width)
        e = -decode<UInt>(-x, n);
      return e;
    }
    x <<= 1;
    if (!msb(x)) {
      e = 0;
```

```
      n = 2;
    }
    else {
      e = 1;
      n = 2;
      x <<= 1;
      // count number of leading ones, m
      int m;
      for (m = 0; m < mmax && msb(x); m++, x <<= 1);
      n += m;
      // shift out terminating zero-bit if m < mmax
      if (m < mmax) {
        n++;
        x <<= 1;
      }
      // decode m residuals
      while (m--) {
        int k = e;
        // decode k-bit residual
        e = (1 << k) + int(x >> (width - k));
        n += k;
        x <<= k;
      }
    }
    return e;
  }

protected:
  // return floor(lg(x))
  static int ilog(unsigned int x)
  {
    int k;
    for (k = 0; x > 1; k++, x >>= 1);
    return k;
  }
};
```

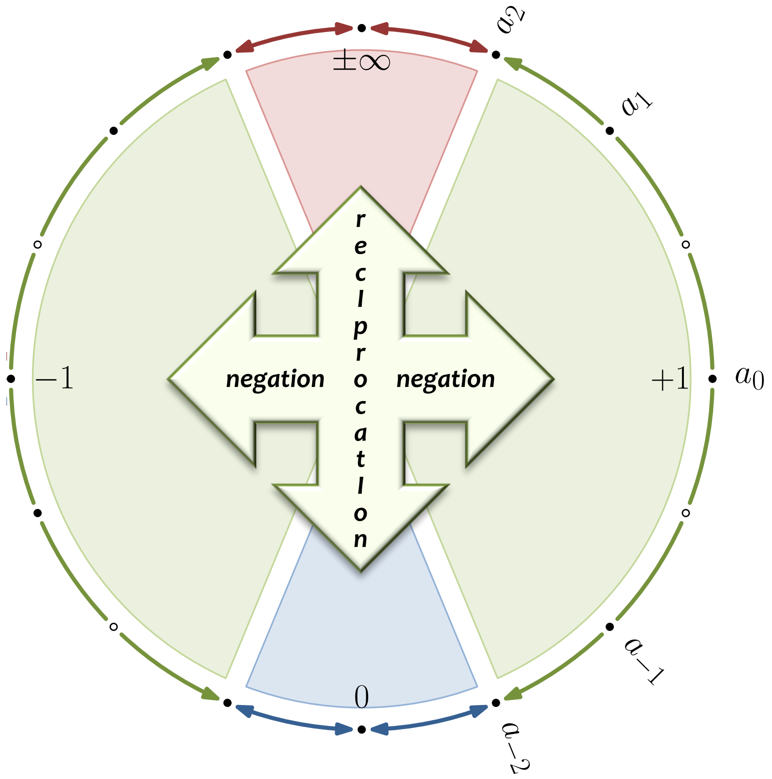
POSITS, ELIAS recursively bisect intervals (0, ±1), (±1, ±∞)
Example: POSIT(0) (aka. ELIAS γ)

- sign bit
- exponent sign
- exponent value
- fraction value

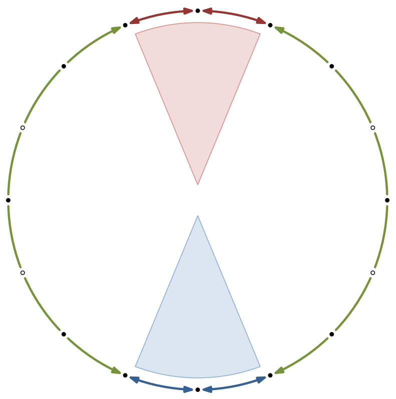


Bisection rule depends on where we are in the ringplot

- bracketing on (x_{max}, ∞)
- refinement on (x_{min}, x_{max})
- reciprocation on $(0, x_{min})$



Case 1 (unbounded search): x lies in interval bounded by ∞ or 0

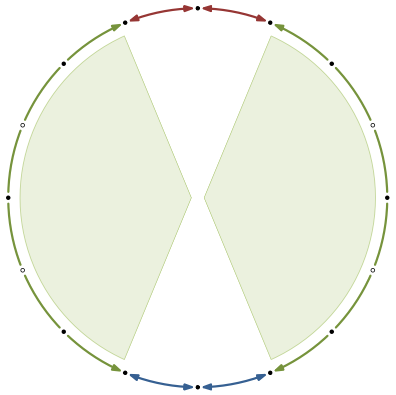


- Unbounded search brackets $x \in [a_i, a_{i+1})$
- $\{a_i\}$ is a monotonically increasing sequence
  - $a_0 = 1$ initial guess
  - $a_i = g(a_{i-1})$ g is a **generator** for the sequence, e.g., $g(x) = 2x$
  - $a_{-i} = 1 / a_i$ generalization of search to $(0, 1)$
- We distinguish three sub-cases
  - $x \in [a_i, a_{i+1}) \subset [1, \infty)$ Code(x) = 01 1^i 0 …
  - $x \in [a_{-(i+1)}, a_{-i}) \subset (0, 1)$ Code(x) = 00 0^i 1 … *reciprocation*
  - $x \in (-\infty, 0)$ Code(x) = –Code(–x) *two's complement negation*

Case 2 (binary search): x has been bracketed in $[a_i, a_{i+1})$

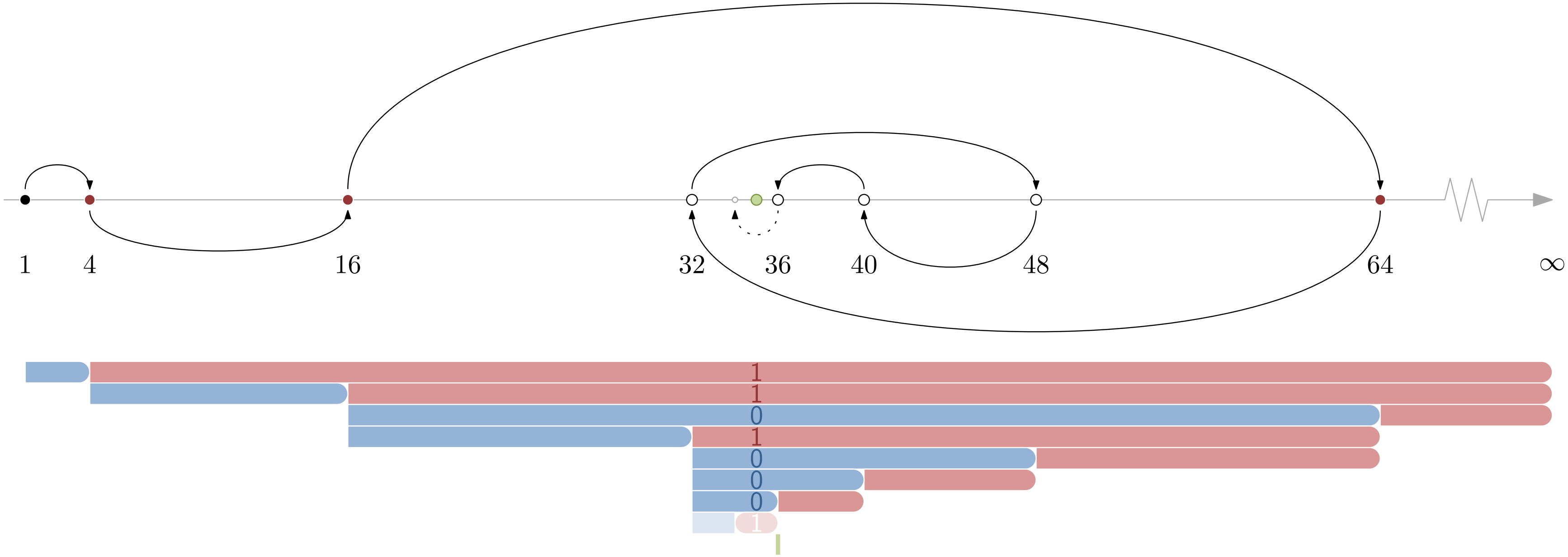
- $c = f(a, b)$ is a **refinement** operator such that $a < c < b$
  - c needs not be the midpoint of (a, b)

- We distinguish two sub-cases
  - $x \in [a, c)$ append 0 to Code(x) and recurse on $[a, c)$
  - $x \in [c, b)$ append 1 to Code(x) and recurse on $[c, b)$



Generator g(x), refinement operator $f(a, b)$ uniquely define the number system

Unbounded & binary search narrow the interval containing x via sequence of comparisons, each of which produces one bit

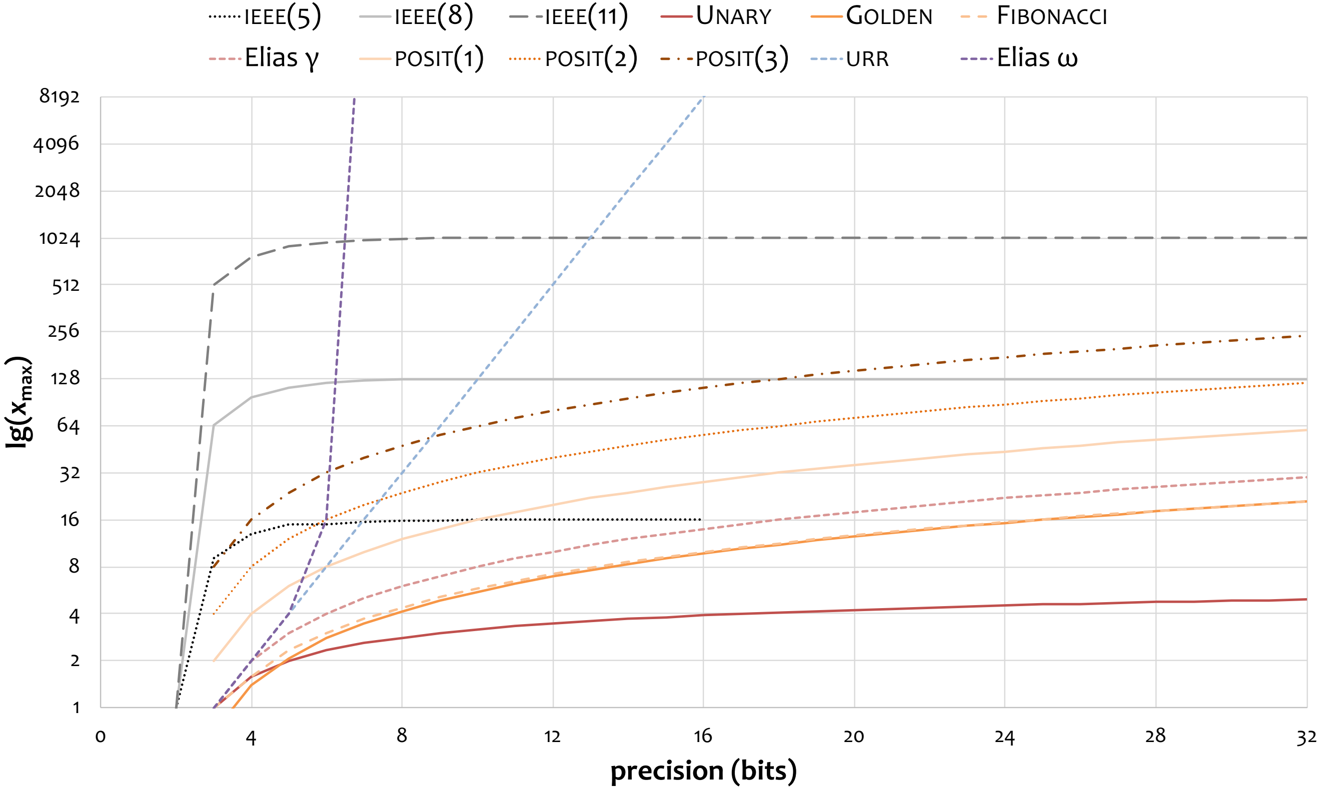


Many known and new representations have simple generators g(x)

type	g(x)	sequence
UNARY	$1 + x$	1, 2, 3, 4, 5, …
Fibonacci	$\mathrm{round}(\phi x)$	1, 2, 3, 5, 8, …
ELIAS γ	$2x$	1, 2, 4, 8, 16, …
POSITS (base *b*)	bx	1, b, b^2, b^3, b^4, …
URR	$\max\{2, x^2\}$	1, 2, 4, 16, 256, …
ELIAS δ	$2x^2$	1, 2, 8, 128, 32768, …
ELIAS ω	2^x	1, 2, 4, 16, 65536, …
IEEE float†	$\sqrt{2^{129}x}$	2, 2^{65}, 2^{97}, 2^{113}, 2^{121}, …

† $g(x) = \frac{2^{128}+x}{2}$ when $x \geq 2^{127}$

Unlike IEEE, universal types have no set ceiling



Refinement operator computes a "mean" of two values

- For POSITS, URR, ELIAS {γ, δ, ω}

$$f(a,b) = \begin{cases} \dfrac{a+b}{2} & \text{if } a = 0 \lor b = 0 \lor b \le 2a \\ 2^{f(\lg a, \lg b)} & \text{otherwise} \end{cases}$$

- Examples
  - $f(2, 4) = 3$ — arithmetic mean
  - $f(4, 16) = 2^{f(2, 4)} = 2^3 = 8$ — geometric mean
  - $f(16, 65536) = 2^{f(4, 16)} = 2^{2^{f(2,4)}} = 2^{2^3} = 2^8 = 256$ — "hyper mean" (for ELIAS ω only)

- For LNS

$$f(a,b) = \sqrt{ab}$$

Natural refinement operator is given by Kolmogorov mean in terms of cumulative distribution function, *F*(*x*)

- Given bracket sequence $\{a_i\}$, how do we find "compatible" refinement operator?

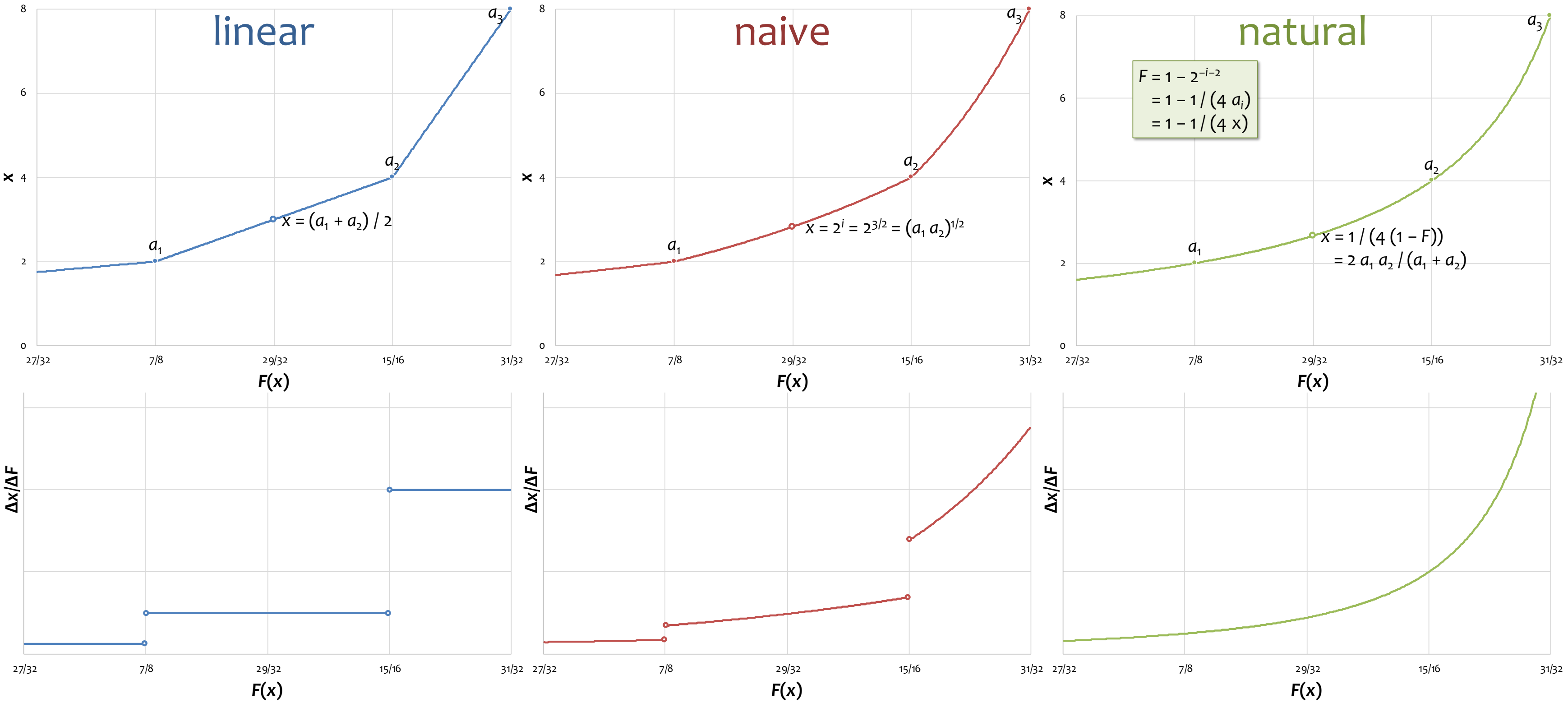
- Solve recurrence $a_i = g(a_{i-1}) = g^i(1) = g \circ g \circ \cdots \circ g(1)$
  - E.g., $a_i = 2a_{i-1} = 2^i$ (ELIAS γ, aka. POSIT(0))

- Solve $x = a_i$ for i
  - E.g., $i = \log_2 x$

- Plug i into $F = 1 - 2^{-i}$
  - E.g., $F = 1 - x^{-1}$

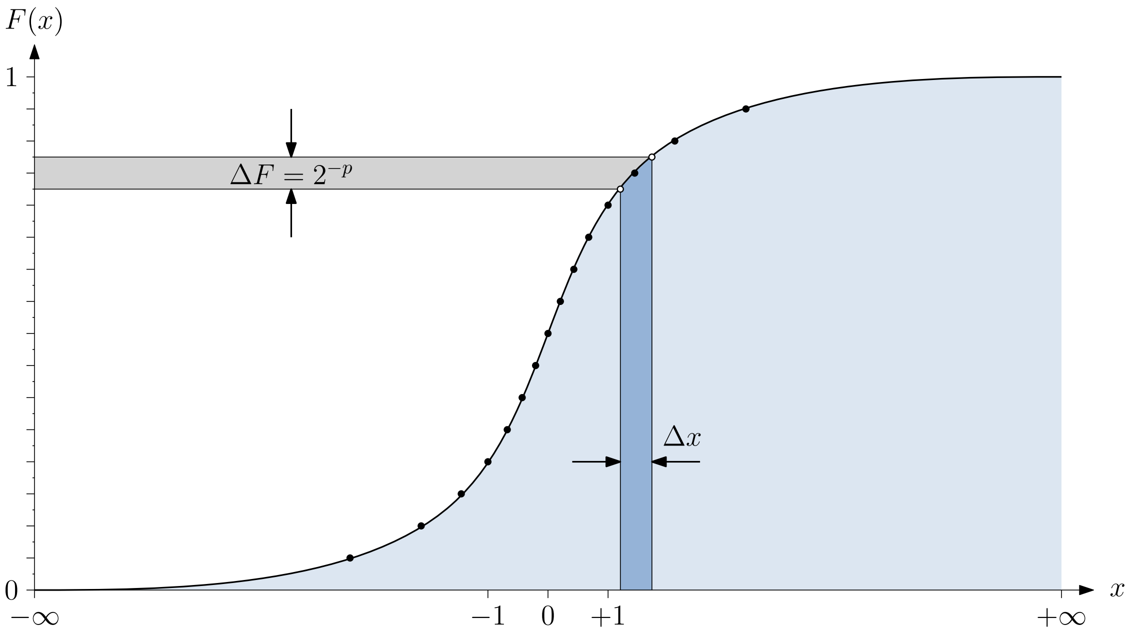
- Plug F into refinement operator $f(a, b) = F^{-1}\left(\frac{F(a)+F(b)}{2}\right)$
  - E.g., $f(a, b) = \frac{2ab}{a+b}$ (harmonic mean)

Natural refinement ensures a smooth and compatible interpolation of the bracket sequence

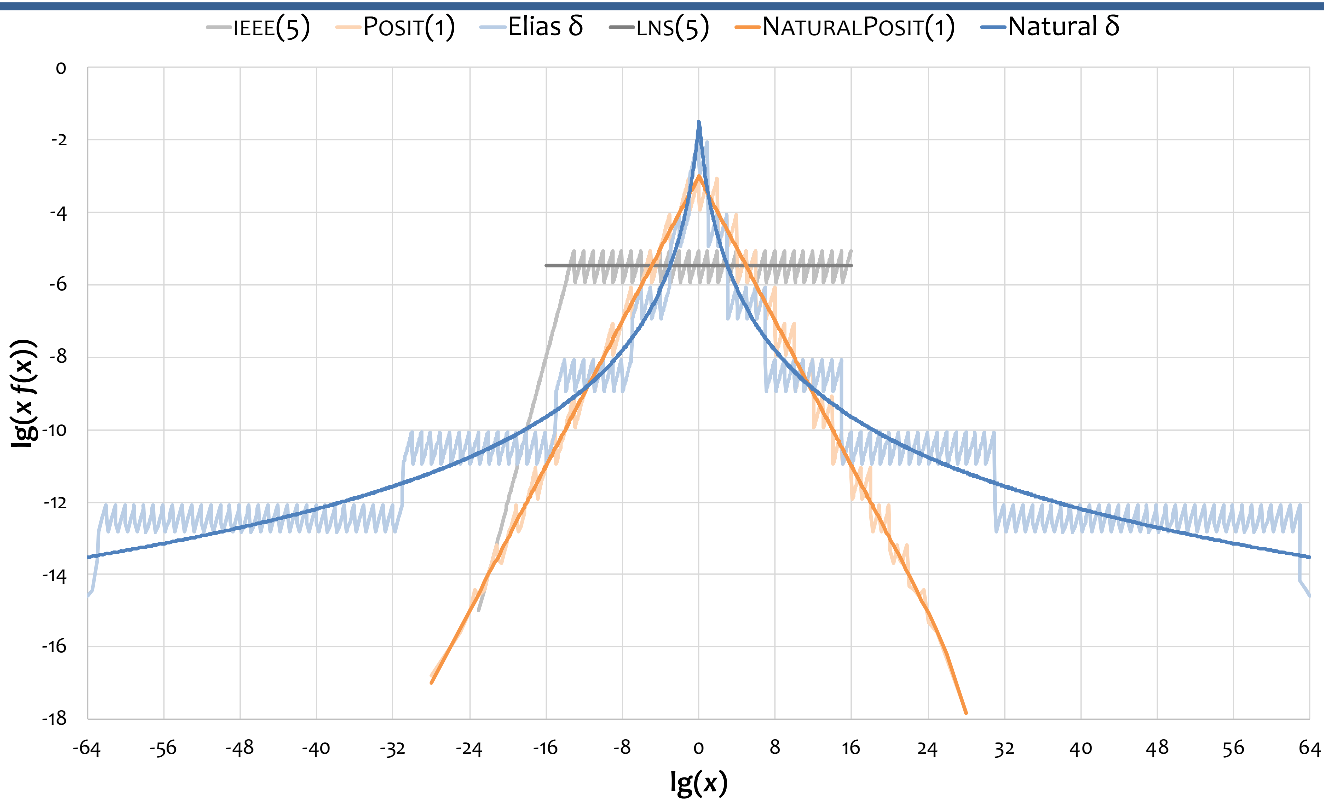


Probability density *f* is connected to relative accuracy α

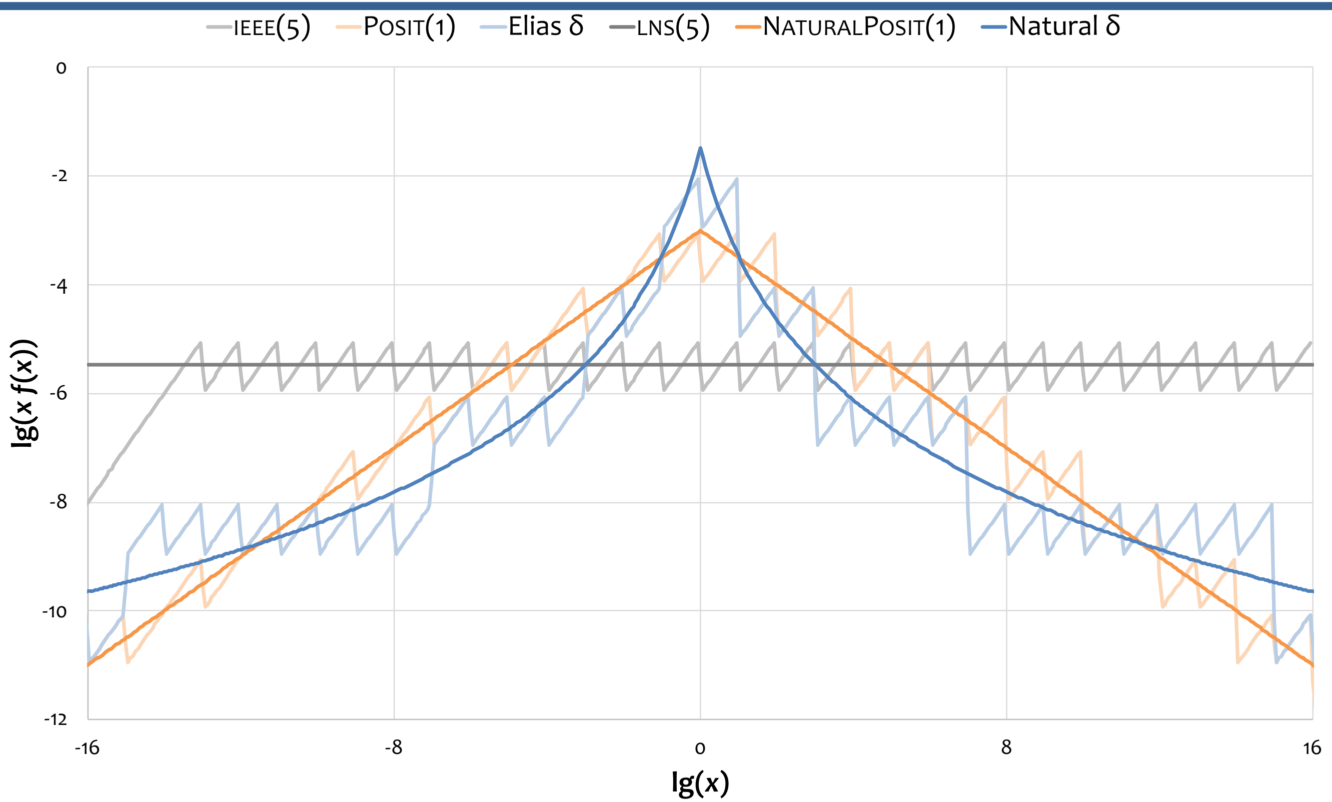
$$\alpha(x) = \lg(xf(x)) = \lg\left(x\frac{dF}{dx}\right) \approx \lg\left(x\frac{\Delta F}{\Delta x}\right) = \lg\left(\frac{x}{\Delta x}2^{-p}\right) = \lg\left(\frac{x}{\Delta x}\right) - p$$



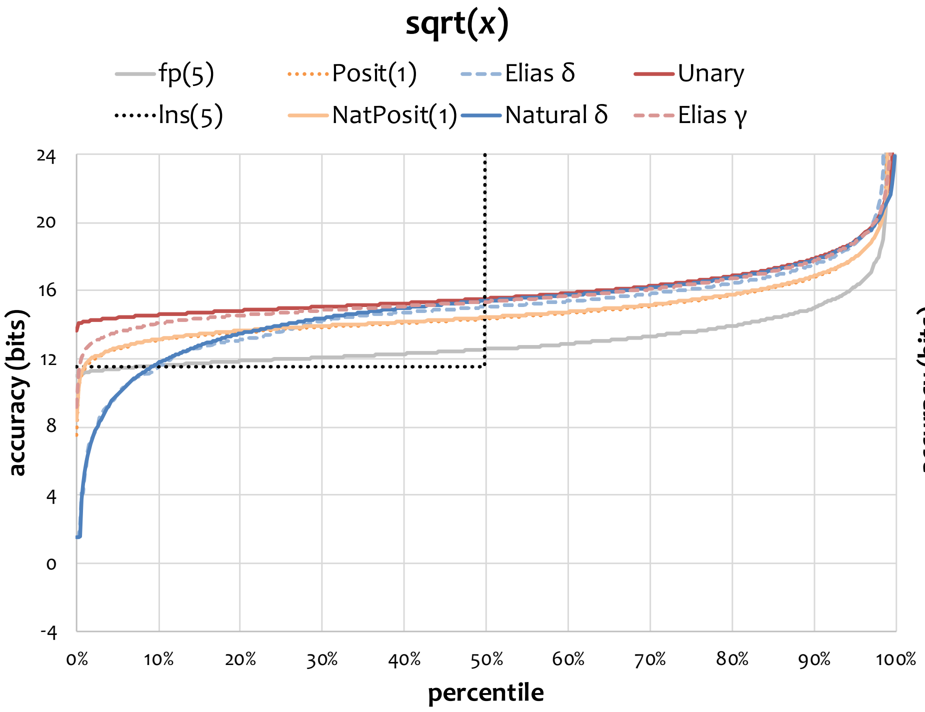
Natural refinement gives smoothly varying accuracy with no wobbles (16-bit precision)

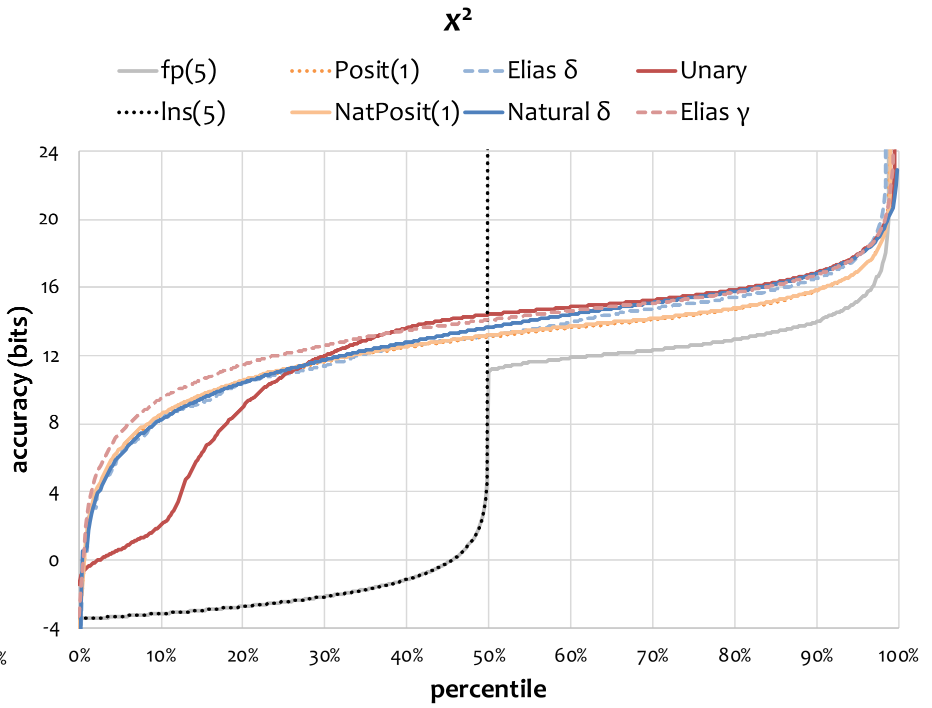


Closeup of accuracy plot reveals sawtooth pattern for piecewise linear refinement

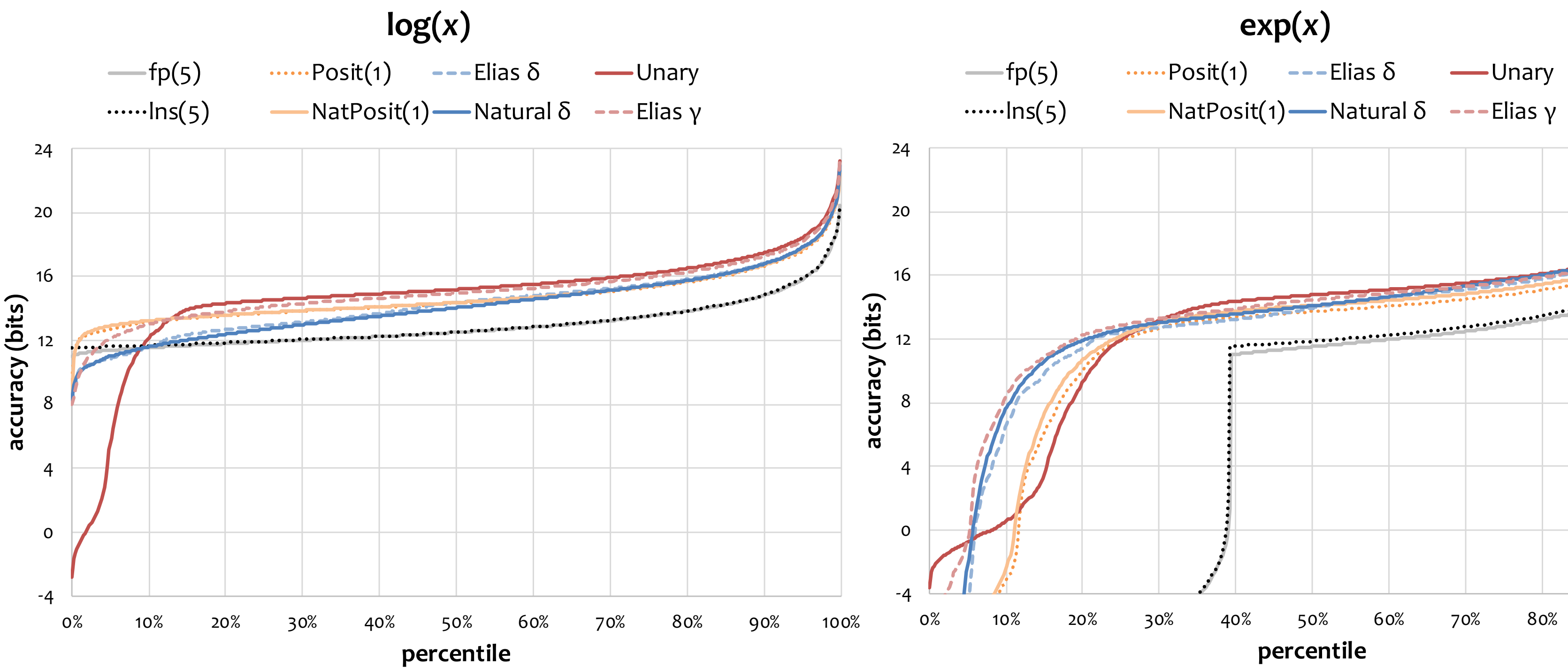


Relative accuracy of unary operators favors universal types (square root and square at 16-bit precision)

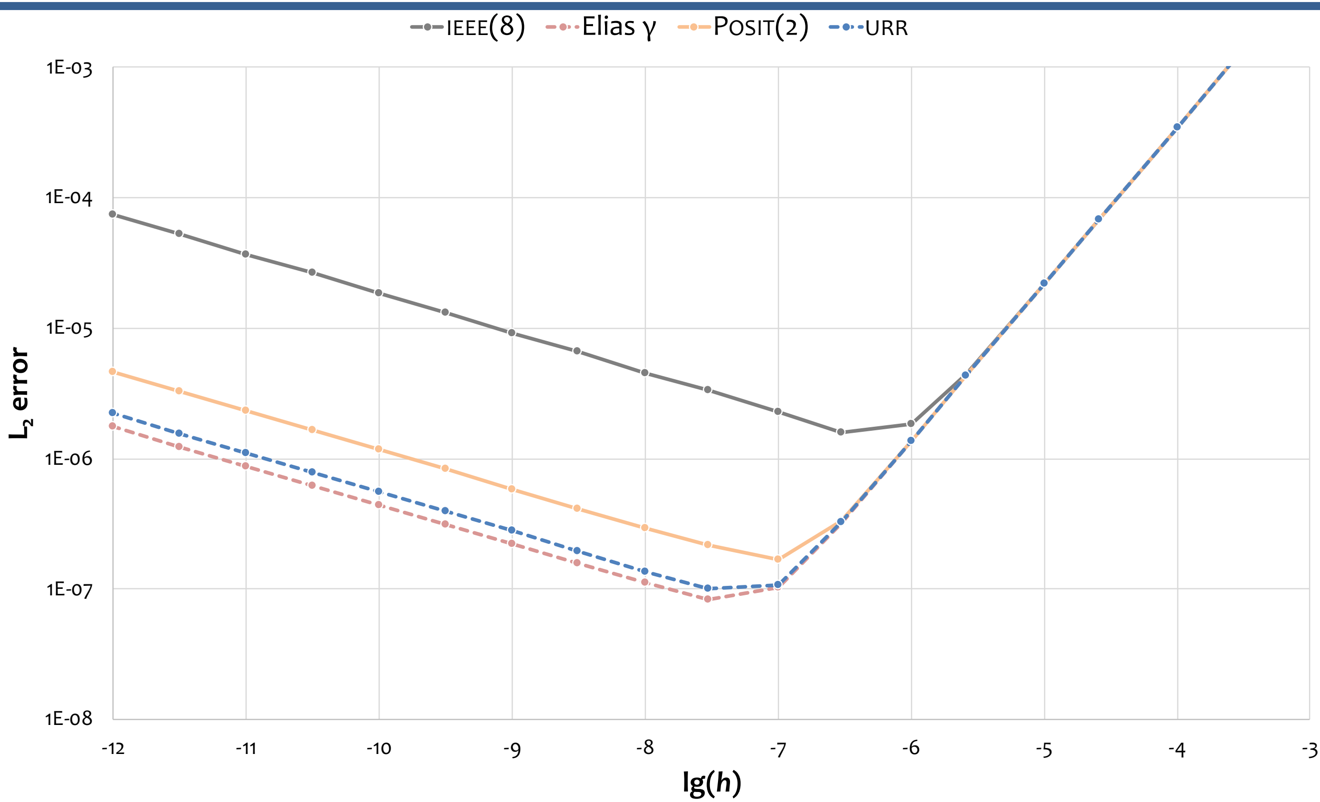




Relative accuracy of unary operators favors universal types (log and exp at 16-bit precision)



Tapered types reduce roundoff error in 32-bit finite differences



Our generic framework is implemented in just two dozen lines of Mathematica code

```
(* decode a binary string, x *)
decode[y_] := decode[StringReplace[y, {"0"->"1", "1"->"0"}, 1], {Indeterminate, Indeterminate}]
decode[y_, {min_, max_}] := decode[StringDrop[y, 1], If[StringTake[y, 1] == "0", {min, bisect[min, max]}, {bisect[min, max], max}]]
decode["", {min_, max_}] := min
(* decode a p-bit signed integer, y *)
decode[y_, p_] := decode[IntegerString[Mod[y, 2^p], 2, p]]


(* encode a real number, x, as a p-bit binary string *)
encode[Indeterminate, p_] := IntegerString[2^(p - 1), 2, p]
encode[0, p_] := IntegerString[0, 2, p]
encode[x_, p_] := IntegerString[BitXor[encode[x, p, {Indeterminate, Indeterminate}, 0], 2^(p - 1)], 2, p]
encode[x_, p_, {min_, max_}, y_] := If[x < bisect[min, max], encode[x, p - 1, {min, bisect[min, max]}, 2 y + 0], encode[x, p - 1, {bisect[min, max], max}, 2 y + 1]]
encode[x_, 0, {min_, max_}, y_] := round[x, {min, max}, y]


(* special rounding rules to avoid under- and overflow *)
round[x_, {Indeterminate, max_}, y_] := y + 1
round[x_, {min_, Indeterminate}, y_] := y + 0
round[x_, {min_, 0}, y_] := y + 0
round[x_, {0, max_}, y_] := y + 1
(* round x to nearest value of {min, max} *)
round[x_, {min_, max_}, y_] := If[x < bisect[min, max], y + 0, If[x > bisect[min, max], y + 1, y + BitAnd[y, 1]]]


(* general bisection rules *)
bisect[Indeterminate, Indeterminate] := 0
bisect[0, Indeterminate] := 1
bisect[min_, max_?NonPositive] := -bisect[-max, -min] (* negation *)
bisect[0, max_] := bisect[max^-1, Indeterminate]^-1 (* reciprocation *)
bisect[min_, Indeterminate] := g[min] (* bracketing *)
bisect[min_, max_] := f[min, max] (* refinement *)


(* power mean and hyper mean *)
pmean[a_, b_, 0] := Sqrt[a b]
pmean[a_, b_, p_] := ((a^p + b^p) / 2)^(1/p)
hmean[a_?Negative, b_?Negative] := -hmean[-a, -b]
hmean[a_, b_] := If[a == 0 || b == 0 || 1/2 <= a / b <= 2, (a + b) / 2, 2^hmean[Log2[a], Log2[b]]]
```

Each number system is a single line of code

```
g[x_] := x + 1;                    f[a_, b_] := 1 - Log2[2^-a + 2^-b]                       (* unary *)
g[x_] := hmean[x, 2^(2^(m-1))];    f[a_, b_] := hmean[a, b]                                  (* FP(m) with m-bit exponent *)
g[x_] := Sqrt[x 2^(2^(m-1))];      f[a_, b_] := Sqrt[a b]                                    (* LNS(m) with m-bit exponent *)
g[x_] := GoldenRatio * x;          f[a_, b_] := pmean[a, b, -1 / Log2[GoldenRatio]]          (* golden ratio *)
g[x_] := Round[GoldenRatio * x];   f[a_, b_] := (a + b) / 2                                  (* Fibonacci *)
g[x_] := (2^m + 1) x;              f[a_, b_] := (a + b) / 2                                  (* base 2^m + 1 *)
g[x_] := 2^m x;                    f[a_, b_] := Sqrt[a b]                                    (* base 2^m *)
g[x_] := 2^m x;                    f[a_, b_] := pmean[a, b, -1 / m]                          (* natural base 2^m *)
g[x_] := 2^(2^m) x;                f[a_, b_] := hmean[a, b]                                  (* posit(m) with m-bit exponent *)
g[x_] := 2^(2^m) x;                f[a_, b_] := pmean[a, b, 2^-m * If[b <= 1, +1, -1]]       (* natural posit(m) *)
g[x_] := Max[2, x^2];              f[a_, b_] := hmean[a, b]                                  (* URR *)
g[x_] := 2 x^2;                    f[a_, b_] := hmean[a, b]                                  (* Elias delta *)
g[x_] := 2 x^2;                    f[a_, b_] := (a^Log2[2 b] b^Log2[2 a])^(1 / Log2[4 a b])  (* natural delta *)
g[x_] := 2^x;                      f[a_, b_] := hmean[a, b]                                  (* Elias omega *)
```

POSIT C++ implementation is 8 lines of code

```
// posit generator with m-bit exponent and base 2^(2^m)
template <int m = 0, typename real>
class Posit {
public:
    real operator()(real x) const { return real(base) * x; }
    real operator()(real x, real y) const { return real(2) * x >= y ? (x + y) / real(2) : std::sqrt(x * y); }
protected:
    static const int base = 1 << (1 << m);
};
```

Our framework is simple and intuitive

- Pros
  - Number system given by two **simple functions**
  - Excellent for **rapid prototyping**
  - Verifiably **correct**
  - Very **general**, e.g., supports exponent-less number systems like UNARY & FIBONACCI
  - Natural refinement ensures **continuous accuracy**
  - **Analytical distribution** enables analysis, e.g., density of sum

- Cons
  - **Inefficent**: Encoding and decoding require *p* steps for *p* bits of precision
    - May specialize/optimize implementation and use out framework to verify its correctness
  - Natural refinement may involve **elaborate expressions** like transcendentals
    - Not hardware friendly, but suitable when accuracy is favored over speed
    - Lookup tables possible for low-precision applications
  - Relies on **auxiliary type** (e.g., IEEE, MPFR) for arithmetic

Conclusions

- Our framework is general and simple to use
  - Implementation of new number system is **two lines of code**
  - Can express most **existing representations**
  - Leads to **novel representations** not expressible in prior frameworks
  - Generator function **intuitively shapes** the number system
- In the limit, bisection leads to a continuous probability distribution
  - CDF-induced **Kolgomorov mean** gives refinement operator for fine-scale shape
  - Probability density gives closed-form expression for **relative accuracy**
- Future work
  - **Evaluate** new number systems in real applications
  - When should we prefer **exponent-less systems, natural refinement**?
  - Is there an **optimal number distribution**?

Lawrence Livermore National Laboratory