Universal Coding of the Reals using Bisection

Conference for Next Generation Arithmetic 2019

March 13, 2019

Peter Lindstrom
I’m thinking of an integer $0 \leq x \leq 99$. 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)

- 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

<table>
<thead>
<tr>
<th></th>
<th>IEEE 754 floating point</th>
<th>POSITS: UNUM version 3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Length</td>
<td>{16, 32, 64, 128}</td>
<td>fixed length but arbitrary</td>
</tr>
<tr>
<td>Sign</td>
<td>sign-magnitude (-0 ≠ +0)</td>
<td>two’s complement (-0 = +0)</td>
</tr>
<tr>
<td>Exponent</td>
<td>fixed length (biased binary)</td>
<td>variable length (Golomb-Rice)</td>
</tr>
<tr>
<td>Fraction Map</td>
<td>linear (φ(x) = 1 + x)</td>
<td>linear (φ(x) = 1 + x)</td>
</tr>
<tr>
<td>Infinities</td>
<td>{-∞, +∞}</td>
<td>±∞ (single point at infinity)</td>
</tr>
<tr>
<td>NaNs</td>
<td>many (9 quadrillion)</td>
<td>one</td>
</tr>
<tr>
<td>Underflow</td>
<td>gradual (subnormals)</td>
<td>gradual (natural)</td>
</tr>
<tr>
<td>Overflow</td>
<td>1 / FLT_TRUE_MIN = ∞ (oops!)</td>
<td>never (exception: 1 / 0 = ±∞)</td>
</tr>
<tr>
<td>Base</td>
<td>{2, 10}</td>
<td>2^m ∈ {2, 4, 16, 256, ...}</td>
</tr>
</tbody>
</table>
In terms of encoding scheme, IEEE and POSITS differ in one important way

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

- **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 \textit{unary}, $m$ trailing bits in \textit{binary} ($e \mod 2^m$)

<table>
<thead>
<tr>
<th>Exponent</th>
<th>IEEE double</th>
<th>POSIT(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01111111111</td>
<td>0000</td>
</tr>
<tr>
<td>13</td>
<td>10000001100</td>
<td>10101</td>
</tr>
<tr>
<td>78</td>
<td>1001001101</td>
<td>111111110</td>
</tr>
</tbody>
</table>
Related work: Universal coding of positive integers [Elias 1975]

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

- Elias proposed three “prefix-free” variable-length codes for integers
  - Elias $\gamma$ code is equivalent to POSIT(0) [Gustafson & Yonemoto 2017]
  - Elias $\delta$ code is essentially equivalent to URR [Hamada 1983]

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Elias $\gamma$ code</th>
<th>Elias $\delta$ code</th>
<th>Elias $\omega$ code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code(1)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Code($2^e + r$)</td>
<td>$1^e \ 0 \ r$</td>
<td>$1 \ \gamma(e) \ r$</td>
<td>$1 \ \omega(e) \ r$</td>
</tr>
<tr>
<td>Code(17 = $2^4 + 1$)</td>
<td>1111 0 0001</td>
<td>111 0 00 0001</td>
<td>1 1 1 0 0 0 0 0 0 0 0 0 1</td>
</tr>
</tbody>
</table>
At CoNGA 2018, we extended ELIAS codes from positive integers to 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)

![Real Number Line with ELIAS Codes](image)
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 \{-\infty, 0, +\infty\}, ...

**overflow & underflow**
- snap to FLT_MAX (FLT_MIN), to infinity (zero), throw exception, ...


NUMRep unifies IEEE, POSITS, ELIAS, URR, LNS, ..., under a single schema
**Elias omega exponent with at most mmax iterated exponentials**

```cpp
public
template
// Elias omega exponent with at most mmax iterated exponentials
template <typename UINT>
static ExpStatus encode(int e, UINT& x, int n);

const int width = CHAR_BIT * sizeof(int);
ExpStatus status = expOK;

if (e < 0) {
  // use encoding of inverted exponent
  status = ExpStatus(-encode<UINT>(e));
  // invert lowest n bits
  x = UINT(~UINT(x)) >> (width - n);
  // correct exponent if it does not fit
  if (status != expOK)
    x = UINT(~x);
  return status;
}

if (e == 0) {
  // e = 0 is encoded as binary 10
  x = 0x2u;
  n = 0;
}
else {
  // e = r(m) + 2^(r(m-1) + 2^(... + 2^k))
  // is encoded as binary 11 1^m (0) r(1) r(2) ... r(m)
  x = 0;
  n = 0;
  int m;
  for (m = 0; m < mmax && e > 1; m++)
    // prepend least k = lg(e) >= 0 bits
    int k = ilog(e);
    int l = width - k;
    if (l > 0) {
      if (((x >> k) << k) != x)
        {
          status = expPosInexact;
          x = k;
          x += UINT(e) << 1;
        }
      else {
        l = -1;
        if (x) {
          status = expPosInexact;
          x = 0;
          if (((UINT(e) >> l) << l) != 0)
            status = expPosInexact;
          x += UINT(e) >> l;

          // zero all but lowest n bits
          e = k;
          n += k;
          // shift out terminating zero-bit if m < mmax
          if (m < mmax)
            n++;  // overflow if there are unprocessed exponent bits
          if (n >= width)
            return status;

          // round based on truncated LSB
          if (i(x) {
            x++;  // prepend terminating zero-bit
            n = width - 1;
          return status;
          }
        }
        // prepend 2 + m one-bits; length mmax
        l = mmax;
        m += 2;
        // prepend least k = lg(e) >= 0 bits
        int k = ilog(e);
        int l = width - k;
        if (l > 0) {
          if (((x >> k) << k) != x)
            {
              status = expPosInexact;
              x = k;
              x += UINT(e) << 1;
            }
          else {
            l = -1;
            if (x) {
              status = expPosInexact;
              x = 0;
              if (((UINT(e) >> l) << l) != 0)
                status = expPosInexact;
              x += UINT(e) >> l;

              // zero all but lowest n bits
              e = k;
              n += k;
            }
          }
        }
        // prepend 2 + m one-bits; length mmax
        l = mmax;
        m += 2;
        // prepend least k = lg(e) >= 0 bits
        int k = ilog(e);
        int l = width - k;
        if (l > 0) {
          if (((x >> k) << k) != x)
            {
              status = expPosInexact;
              x = k;
              x += UINT(e) << 1;
            }
          else {
            l = -1;
            if (x) {
              status = expPosInexact;
              x = 0;
              if (((UINT(e) >> l) << l) != 0)
                status = expPosInexact;
              x += UINT(e) >> l;

              // zero all but lowest n bits
              e = k;
              n += k;
            }
          }
        }
        // prepend 2 + m one-bits; length mmax
        l = mmax;
        m += 2;
        // prepend least k = lg(e) >= 0 bits
        int k = ilog(e);
        int l = width - k;
        if (l > 0) {
          if (((x >> k) << k) != x)
            {
              status = expPosInexact;
              x = k;
              x += UINT(e) << 1;
            }
          else {
            l = -1;
            if (x) {
              status = expPosInexact;
              x = 0;
              if (((UINT(e) >> l) << l) != 0)
                status = expPosInexact;
              x += UINT(e) >> l;

              // zero all but lowest n bits
              e = k;
              n += k;
            }
          }
        }
    }
}

// decode m residuals
while (m--) {
  int k = e;
  // decode k-bit residual
  e = ((1 << k) + int(x) >> (width - k));
  n += k;
  x <<= 1;
}

// decode n residues
while (n--) {
  int k = e;
  // decode k-bit residual
  e = ((1 << k) + int(x) >> (width - k));
  n += k;
  x <<= 1;
}

protected:
  // return floor(lg(x))
  static int ilog(unsigned int x) {
    int k;
    for (k = 0; x > 1; k++, x >>= 1);  // overflow if there are unprocessed exponent bits
    return status;
  }
}
```

---

**NumRep implementation can be tricky and error prone:**

Elias ω exponent coding alone is 144 lines of C++ code.
**Example:** \texttt{POSIT(0)} (aka. \texttt{ELIAS} $\gamma$)
Bisection rule depends on where we are in the ringplot

- bracketing on \((x_{\text{max}}, \infty)\)
- refinement on \((x_{\text{min}}, x_{\text{max}})\)
- reciprocation on \((0, x_{\text{min}})\)
Case 1 (unbounded search): $x$ lies in interval bounded by $\infty$ 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
  - $x \in [a_i, a_{i+1}) \subseteq [1, \infty)$ $\text{Code}(x) = 01^i 0 \ldots$
  - $x \in [a_{-(i+1)}, a_{-i}) \subseteq (0, 1)$ $\text{Code}(x) = 00^i 1 \ldots$ reciprocation
  - $x \in (-\infty, 0)$ $\text{Code}(x) = \text{Code}(-x)$ 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 $\text{Code}(x)$ and recurse on $[a, c)$
  - $x \in [c, b)$ append 1 to $\text{Code}(x)$ 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 $g(x)$

<table>
<thead>
<tr>
<th>type</th>
<th>$g(x)$</th>
<th>sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UNARY</strong></td>
<td>$1 + x$</td>
<td>$1, 2, 3, 4, 5, \ldots$</td>
</tr>
<tr>
<td><strong>Fibonacci</strong></td>
<td>$\text{round}(\phi x)$</td>
<td>$1, 2, 3, 5, 8, \ldots$</td>
</tr>
<tr>
<td><strong>ELIAS $\gamma$</strong></td>
<td>$2x$</td>
<td>$1, 2, 4, 8, 16, \ldots$</td>
</tr>
<tr>
<td><strong>POSITS (base $b$)</strong></td>
<td>$bx$</td>
<td>$1, b, b^2, b^3, b^4, \ldots$</td>
</tr>
<tr>
<td><strong>URR</strong></td>
<td>$\max{2, x^2}$</td>
<td>$1, 2, 4, 16, 256, \ldots$</td>
</tr>
<tr>
<td><strong>ELIAS $\delta$</strong></td>
<td>$2x^2$</td>
<td>$1, 2, 8, 128, 32768, \ldots$</td>
</tr>
<tr>
<td><strong>ELIAS $\omega$</strong></td>
<td>$2^x$</td>
<td>$1, 2, 4, 16, 65536, \ldots$</td>
</tr>
<tr>
<td><strong>IEEE float</strong></td>
<td>$\sqrt{2^{129}} x$</td>
<td>$2, 2^{65}, 2^{97}, 2^{113}, 2^{121}, \ldots$</td>
</tr>
</tbody>
</table>

$g(x) = \frac{2^{128} + x}{2}$ 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** \( \{\gamma, \delta, \omega\} \)

\[
f(a, b) = \begin{cases} 
\frac{a + b}{2} & \text{if } a = 0 \lor b = 0 \lor b \leq 2a \\
2^{f(\lg a, \lg b)} & \text{otherwise}
\end{cases}
\]

- **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 \( \omega \) 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 $a_i = g(a_{i-1}) = g^i(1) = g \circ g \circ \ldots \circ g(1)$
    - 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

- **Linear**: \( x = (a_i + a_{i+1}) / 2 \)
- **Naive**: \( x = 2^i \cdot 2^{3/2} = (a_i a_{i+1})^{3/2} \)
- **Natural**: \( x = 1 / (4 (1 - F)) = 1 - 1 / (4 x) \)

**Equations**:
- Linear: \( F = 1 - 2^{1-x} \)
- Naive: \( F = 1 - 1 / (4 a) \)
- Natural: \( F = 1 - 1 / (4 x) \)
Probability density $f$ is connected to relative accuracy $\alpha$

$$\alpha(x) = \lg(xf(x)) = \lg\left(x \frac{dF}{dx}\right) \approx \lg\left(x \frac{\Delta F}{\Delta x}\right) = \lg\left(\frac{x}{\Delta x} 2^{-p}\right) = \lg\left(\frac{x}{\Delta x}\right) - p$$
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_, {min, max}_] := If[x =!= Indeterminate, encode[x, p - 1, {min, bisect[min, max]}], 2 y + 0]
encode[x_, p_, {min, max}_] := 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 =!= Indeterminate, encode[x, p - 1, {min, bisect[min, max]}], 2 y + 0]
round[x_, {min, max}, y_] := round[x, min, max]

(* 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] (* reciprocation *)
bisect[min_, Indeterminate] := g[min] (* bracketing *)
bisect[min_, max_] := f[min, max] (* refinement *)

(* power mean and hyper mean *)
mean[a_, b_, 0] := Sqrt[a b]
mean[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

```
Each number system is a single line of code

\[ g[x_] := x + 1; \quad f[a_, b_] := 1 - \log_2[2^a + 2^b] \quad (* \text{unary} *) \]

\[ g[x_] := \text{hmean}[x, 2^{(2^m-1)}]; \quad f[a_, b_] := \text{hmean}[a, b] \quad (* \text{FP(m) with m-bit exponent} *) \]

\[ g[x_] := \text{Sqrt}[x 2^{(2^m-1)}]; \quad f[a_, b_] := \text{Sqrt}[a b] \quad (* \text{LNS(m) with m-bit exponent} *) \]

\[ g[x_] := \text{GoldenRatio} \times x; \quad f[a_, b_] := \text{pmean}[a, b, -1 / \log_2[\text{GoldenRatio}]] \quad (* \text{golden ratio} *) \]

\[ g[x_] := \text{Round}[\text{GoldenRatio} \times x]; \quad f[a_, b_] := (a + b) / 2 \quad (* \text{Fibonacci} *) \]

\[ g[x_] := (2^m + 1) x; \quad f[a_, b_] := (a + b) / 2 \quad (* \text{base} 2^m + 1 *) \]

\[ g[x_] := 2^m x; \quad f[a_, b_] := \text{Sqrt}[a b] \quad (* \text{base} 2^m *) \]

\[ g[x_] := 2^{(2^m)} x; \quad f[a_, b_] := \text{hmean}[a, b] \quad (* \text{natural base} 2^m *) \]

\[ g[x_] := 2^{(2^m)} x; \quad f[a_, b_] := \text{pmean}[a, b, 2^{-m} * \text{If}[b \leq 1, +1, -1]] \quad (* \text{natural posit(m)} *) \]

\[ g[x_] := \text{Max}[2, x^2]; \quad f[a_, b_] := \text{hmean}[a, b] \quad (* \text{URR} *) \]

\[ g[x_] := 2 \times^2; \quad f[a_, b_] := \text{hmean}[a, b] \quad (* \text{Elias delta} *) \]

\[ g[x_] := 2 \times^2; \quad f[a_, b_] := (a \times \log_2[2 b] \times \log_2[2 a])/(1 / \log_2[4 a b]) \quad (* \text{natural delta} *) \]

\[ g[x_] := 2^x; \quad f[a_, b_] := \text{hmean}[a, b] \quad (* \text{Elias omega} *) \]
```
// 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**
  - Inefficient: 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?