Chapter 3: Refinement and Capacity Increase

Probabilistic Minds Consortium (voids.blog)

2024/2025

1 Introduction

In a finite-capacity (grains-coded) system, each observer state a has a denominator $N(a)$. Sometimes, you need more precision than your current $N(a)$ allows—say, to approximate $\sqrt{2}$ more accurately or to reduce your grains-coded error below a desired threshold. In such cases, you refine capacity, moving from the current representation $\frac{k}{N(a)}$ to a finer one $\frac{k'}{N(a')}$ where $N(a')$ is larger.

In this section, we explore:

- The need for refinement and its role as a bridge between theoretical precision and practical computation.
- How increasing the denominator $N(a)$ (i.e., refining capacity) results in finer increments.
- The process of exact embedding, ensuring that old values are preserved without rounding errors.
- The consistency of grains-coded logic, maintaining all operations as exact integer scalings.

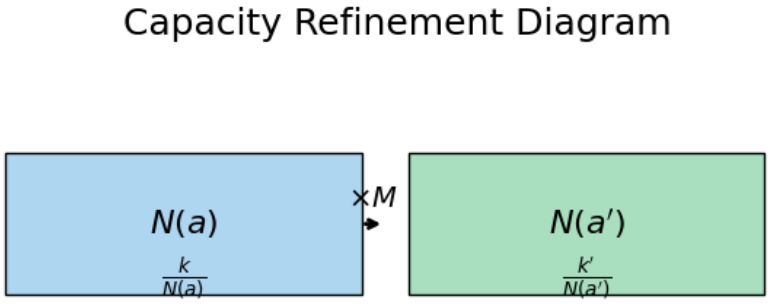
2 Why Do We Need Refinement?

Refinement represents the bridge between theoretical precision and practical computation. In finite mathematics, our numbers are expressed as grains-coded fractions

$\frac{k}{N(a)}$. When the current precision level (determined by $N(a)$) is insufficient—say, when approximating a value with an error of 0.01, and you need to reduce this to 0.001—you must refine capacity.

Key Points:

1. **Larger Denominator = Finer Increments:** Increasing $N(a)$ by refining capacity means that the smallest representable increment $\frac{1}{N(a)}$ becomes smaller.
2. **Exact Embedding: No Rounding Required:** If you choose an integer multiple (for example, if $N(a) = 10$ and you expand by an integer factor $M = 5$, then $N(a') = 5 \times 10 = 50$), the new representation $\frac{k'}{N(a')}$ is exactly equivalent to the old $\frac{k}{N(a)}$. This is critical for preserving computed values.
3. **Consistent with Grains-Coded Logic:** The process is purely an integer scaling of numerators; thus, the overall system remains strictly finite and integer-based.



3 Formalizing Capacity Refinement

3.1 The Embedding Equation

When transitioning from capacity $N(a)$ to capacity $N(a')$, we define an integer expansion factor M such that:

$$N(a') = M \cdot N(a).$$

For any original fraction $\frac{k}{N(a)}$, the value is embedded in the new capacity by setting:

$$\frac{k}{N(a)} \mapsto \frac{k'}{N(a')} \quad \text{where} \quad k' = k \cdot M.$$

This equation ensures:

1. **No Rounding:**
$$\frac{k'}{N(a')} = \frac{k \cdot M}{M \cdot N(a)} = \frac{k}{N(a)},$$
preserving the value exactly.

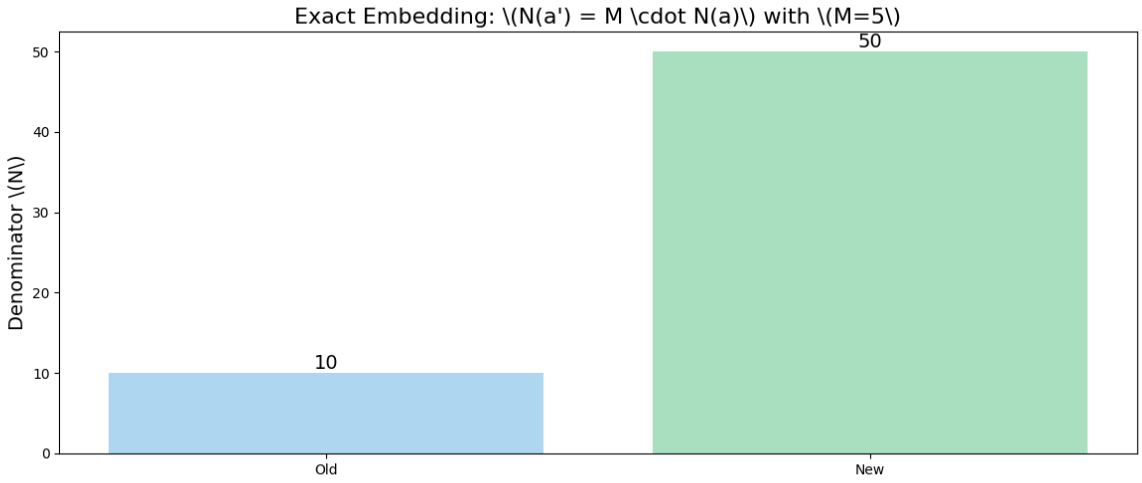
2. **Finite Step:** Every refinement is performed using exact, integer-based operations, ensuring that the grains-coded framework remains finite.

Clarification: The embedding equation is not merely mathematical—it embodies our principle that precision can be increased without loss. By requiring the scaling factor M to be an integer, every refinement preserves computed values while allowing for finer resolution. (See the implementation in `grain_arithmetic.py` for a practical demonstration.)

3.2 Axiom for Refinement

More formally, we require that if an observer state a' is considered "higher capacity" than a, then:
$$N(a') \geq M \cdot N(a)$$
for some integer $M \geq 1$. Typically, M is chosen as an exact integer multiple so that the old fractions embed exactly into the new scale.

Clarification: This axiom explains how probability increments are scaled when capacity increases. In code, when a capacity upgrade occurs, you multiply the old denominator by M to obtain the new denominator, ensuring consistency.



Code Example: `embed_grains_fraction(...)`

```
def embed_grains_fraction(old_k, old_N, new_N):
    """
    Embed (old_k / old_N) into a finer vantage with denominator new_N.
    Assumes new_N = M * old_N for some integer M.
    """
    if new_N % old_N != 0:
        raise ValueError(f"new_N={new_N} not a multiple of old_N={old_N}.")
    factor = new_N // old_N
    new_k = old_k * factor
    return new_k, new_N
```

Explanation:

- If the current grains-coded error is above a certain threshold, and additional expansion is possible, the system multiplies $N(a)$ by an integer factor M (the EXPANSION_FACTOR) and rescales k accordingly.
- This guarantees that the new fraction $\frac{k'}{N(a')}$ is exactly equivalent to the old one, achieving finer granularity without rounding.

Example: If you have $\frac{3}{10}$ and expand to a new capacity of 50 (with $M = 5$), then $\frac{3}{10}$ becomes $\frac{15}{50}$.

4 Refinement in Action: Increasing Precision

4.1 Case Study: Approximating Square Roots

A common grains-coded scenario is approximating a square root with discrete fractions. Suppose $x = \frac{k}{M}$ approximates $\sqrt{N}$. If the error $|x^2 - N|$ is too large, the system refines capacity by increasing M (and adjusting k accordingly) until the approximation is acceptable.

Clarification: This process ensures that the value $\frac{k}{M}$ remains exactly the same when embedded into a finer capacity, while the resolution increases.

4.2 Example Code: `approx_sqrtN_grains(...)`

```
def approx_sqrtN_grains(N_val, INITIAL_K=14, INITIAL_M=10,
                        MAX_CAPACITY=200000, EXPANSION_FACTOR=10):
    k, M = INITIAL_K, INITIAL_M
    while True:
        err = grains_error(k, M, N_val)  # e.g., compare (k/M)^2 vs N_val
```

```
        if err > 0 and M < MAX_CAPACITY:
            newM = M * EXPANSION_FACTOR
            # Embed the old fraction into the new capacity:
            k = int(round(k * newM / M))
            M = newM
        else:
            break
    return k, M
```

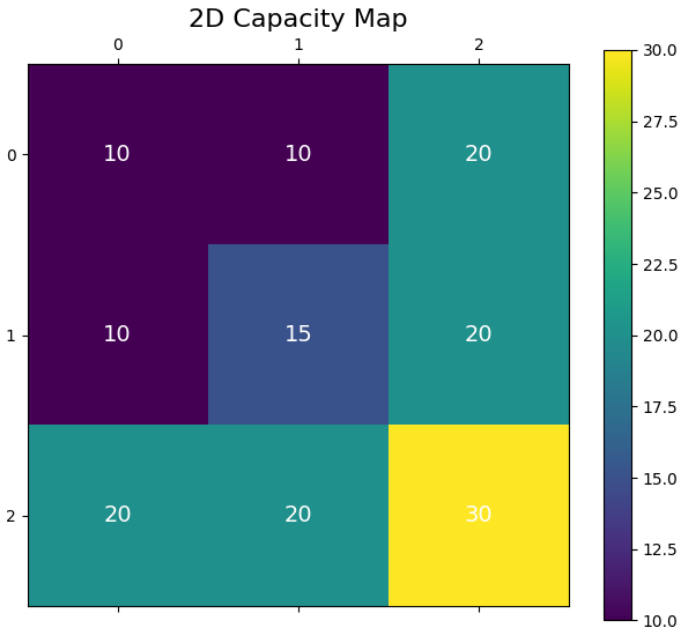
Explanation:

- If the current grains-coded error exceeds a threshold, and further expansion is possible, the capacity M is increased by a factor (EXPANSION_FACTOR).
- The numerator k is rescaled accordingly, ensuring that $\frac{k}{M}$ remains exactly the same, now with finer granularity.

Example: For approximating $\sqrt{2}$, starting with $\frac{14}{10} = 1.4$, iterative refinement may eventually yield $\frac{1414}{1000} \approx 1.414$.

4.3 Block-Based Probability Management: A 2D Implementation

In more complex state spaces, such as a two-dimensional grid, each cell (or block) represents a local probability distribution. Refinement is applied selectively—only in regions where the grains-coded error exceeds a threshold.



Clarification: This adaptive, block-based management allows for targeted precision enhancements without uniformly increasing capacity everywhere, preserving computational resources. For instance, a script like `quantum_check_final.py` demonstrates how spatial locality can be exploited for probability evolution.

5 Ensuring Consistency Across Refinements

Error Monotonicity Axiom

When refining from capacity $N(a)$ to $N(a')$ (with $N(a') = M \cdot N(a)$), the exact embedding ensures that no new rounding errors are introduced. Formally, we require:

$$\forall\, x: \quad \mathrm{Error}\big(x, N(a')\big) \leq \mathrm{Error}\big(x, N(a)\big),$$

where $\mathrm{Error}(x, N)$ denotes the grains-coded error for representing x with capacity N.

Interpretation: Expanding capacity should preserve or reduce error, enabling us to approach the desired precision without degradation.

Practical Takeaways:

- Increase capacity when the current error exceeds a threshold.
- Always choose an integer multiple M so that old fractions embed exactly into the new scale.
- Avoid over-refinement; large denominators can slow computations.
- Using integer expansions exclusively prevents any floating-point rounding errors.

6 What to Try in Code?

1. **Basic Fraction Embedding:** Test the function `embed_grains_fraction(old_k, old_N, new_N)` with small expansions (e.g., from 10 to 50) to observe exact scaling.
2. **Square Root Example:** Run `approx_sqrtN_grains` to observe dynamic capacity refinement (e.g., $\frac{14}{10} \to \frac{140}{100} \to \ldots$) and track error reduction.
3. **Error Tracking:** Modify the function `grains_error` to log error decreases as capacity refines, providing insight into convergence.

7 Conclusion for Section 3

Refinement is the process by which you upgrade from one grains-coded vantage to a "finer" one, preserving all previously computed values through exact integer scaling. This method maintains the entire system as finite and integer-based, while flexibly handling high-precision tasks. The techniques described—exact embedding, capacity expansion, and error monotonicity—demonstrate how the system can dynamically adapt its precision without compromising its discrete, grains-coded nature.

Next, in Section 4, you will see how these refinements feed into grains-coded arithmetic for building a finite analog to classical operations (such as addition and multiplication), all while remaining completely discrete and consistent.