Chapter 4: Emergence of Numbers and Arithmetic

Probabilistic Minds Consortium (voids.blog)

2024/2025

1 Introduction

At first, grains-coded systems may seem limited to storing discrete probabilities or distances. However, all classical arithmetic can emerge from these finite increments. By carefully unifying denominators (denoted $N(a)$) and embedding old fractions into new vantage states, one can replicate addition, multiplication, and even construct rational intervals—all without resorting to infinite sets or continuous real numbers. In essence, what appears to be the domain of continuous numbers is recovered step-by-step through purely discrete operations.

2 Constructing Increment-Based Numbers

The emergence of number systems from our grains-coded framework demonstrates a profound principle: it is possible to construct all necessary mathematical operations using only finite, discrete steps. This not only provides a theoretical alternative to classical real numbers but also supplies a practical foundation for computation. The approach maintains mathematical rigor while eliminating the ambiguities and pitfalls often associated with floating-point arithmetic.

2.1 Finite Intervals from Pattern Distances

Suppose there exist two stable patterns x and y at a given observer state a. Let L be an integer indicating that x and y differ by exactly L minimal increments (as defined in

earlier sections). We can then interpret these L increments as a segment of length 1 in the grains-coded "distance space" at (a, e).

Interval Definition: We define an interval between x and y as the finite set of fractions

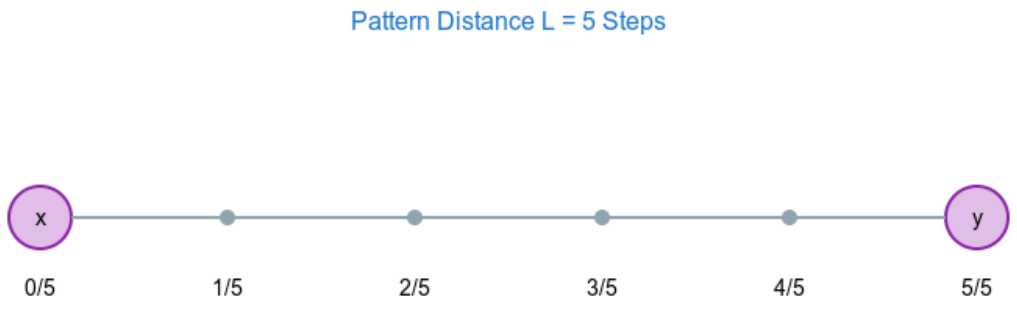
$$\left\{0,\ \frac{1}{L},\ \frac{2}{L},\ \ldots,\ 1\right\}.$$

Each fraction $\frac{k}{L}$ represents a discrete step from x to y.

Finite Fractions: These fractions capture the idea that if you can move from x to y in L single steps, then each step covers exactly $\frac{1}{L}$ of the distance. No continuous continuum is needed here—only the discrete increments, making the method both implementable and exact.

Extended Explanation: This construction shows that a line segment does not require an infinitely divisible continuum. Instead, it is composed of a fixed number of "grains" or increments. In practical programming terms, you could represent the interval as an array: $[0, \frac{1}{L}, \frac{2}{L}, \ldots, 1]$. This representation is exact and supports further arithmetic operations without loss of precision.

IT Analogy: Consider a progress bar that increments in 10% steps. The bar displays a finite, precise sequence (0%, 10%, ..., 100%). Similarly, our interval is a set of precise fractions that exactly partition the segment between two patterns.



2.2 Rational Numbers from Probability Increments

The finite sets

$$\{0,\ \tfrac{1}{L},\ \tfrac{2}{L},\ \ldots, 1\}$$

are effectively the same as the discrete rational increments used to express grains-coded probabilities like $\frac{k}{N(a)}$. Each observer state's capacity, given by $N(a)$, provides a finite set of denominators from which these fractions are built. Rather than forming a continuous infinity, you only obtain multiples of $\frac{1}{N(a)}$.

Why This Matters:

- Every time you define a new capacity $N(a')$ with a larger denominator, the set of representable fractions becomes denser.
- Yet each observer state remains finite, ensuring that the system is free from unbounded or irrational expansions.

Example: If $N(a) = 10$, the set is $\{0, \frac{1}{10}, \frac{2}{10}, \ldots, 1\}$. Refining to $N(a') = 100$ lets you represent fractions as $\frac{k}{100}$. Each state is purely discrete, with no infinite set emerging.

2.3 Operations on Increment-Derived Numbers

2.3.1 Addition as Common Denominator

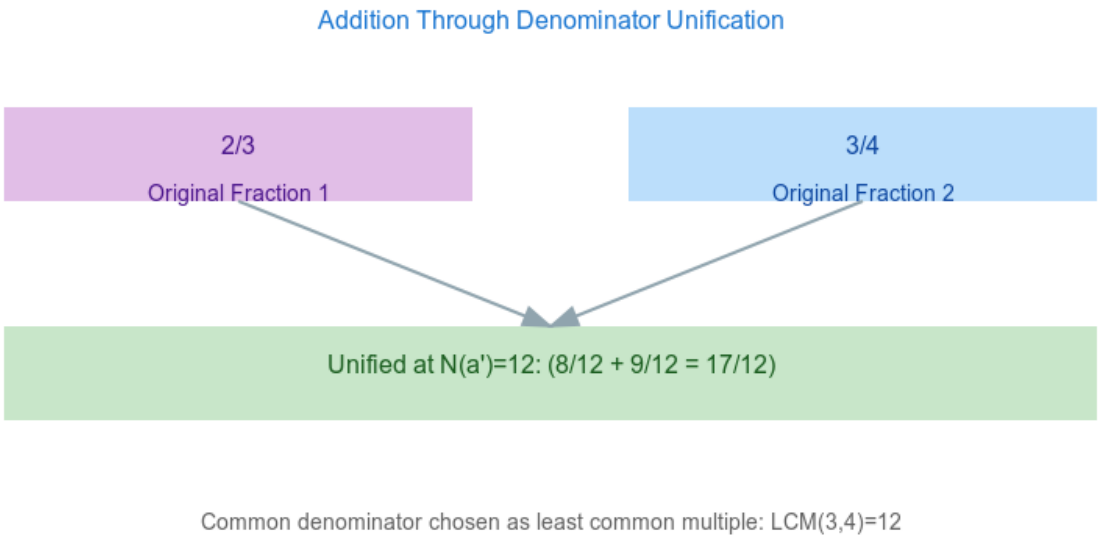
Suppose you have two grains-coded fractions, $\frac{k}{L}$ and $\frac{m}{M}$, possibly from different observer states or intervals. To add them:

1. **Refinement:** Choose a new observer state a' with a capacity $N(a')$ large enough to accommodate both denominators L and M.
2. **Embed:** Convert the fractions $\frac{k}{L}$ and $\frac{m}{M}$ into fractions with the common denominator $N(a')$.
3. **Combine:** Once both fractions are expressed with denominator $N(a')$, add them:

$$\frac{k'}{N(a')} + \frac{m'}{N(a')} = \frac{k' + m'}{N(a')}.$$

Extended Explanation: The key point is that all arithmetic is carried out on fractions with a common denominator. This mirrors the standard method of adding fractions in elementary arithmetic. By "embedding" the original fractions into a refined capacity, we ensure that their sum is computed exactly without any rounding errors.

IT Analogy: Imagine having two data sets stored with different precision levels. To combine them accurately, you first convert both to the highest available precision and then add them. Here, that highest precision is represented by the refined capacity $N(a')$.



2.3.2 No Need for Infinity

All numeric operations remain in finite sets of rationals:

- Each step is computed using integer arithmetic: choose a refined capacity a', scale numerators accordingly, and produce new fractions.
- You never need to work with an unbounded set of denominators—only a single, sufficiently large capacity.

Result: By restricting each observer state to a finite set $\{0, \frac{1}{N(a')}, \ldots, 1\}$, classical fraction arithmetic is replicated exactly in a grains-coded system.

Key Point: All arithmetic operations (addition, subtraction, multiplication, division) are performed exactly with finite, discrete values. No appeal to infinite series or limits is necessary, ensuring that every computation remains precise and computationally feasible.

2.4 No Infinite Sets, No True Irrationals

2.4.1 Only Finite Rationals

Within any single observer state a, the set of representable numbers is:

$$\left\{ \frac{0}{N(a)}, \frac{1}{N(a)}, \ldots, \frac{N(a)}{N(a)} \right\}.$$

This is a finite set. Any arithmetic operation produces results in the form $\frac{k'}{N(a')}$ for some refined observer state a'. Thus, you never leave the realm of finite, integer-based fractions.

2.4.2 Approximation of Finer Values

As the capacity $N(a)$ increases, a denser set of rational increments is available. Although the system remains finite at every stage, it can approximate a "limit" in classical mathematics through stepwise refinements:

- **Stepwise Refinement:** You approximate continuous behavior by progressively increasing $N(a)$ (e.g., from $\frac{14}{10}$ to $\frac{141}{100}$ to $\frac{1414}{1000}$, etc.).
- **Finite Grids:** Each refinement provides a finer, yet still finite, grid for representing numbers.

Extended Explanation: The process of refinement is analogous to zooming in on a digital image: the resolution increases, but the image remains a finite collection of pixels. Similarly, by increasing the denominator, you obtain a more refined approximation of a number while always staying within a finite system.

IT Analogy: Consider a high-resolution display versus a low-resolution one. The high-resolution screen can render smoother curves due to the greater number of pixels, yet it remains a finite array. Our refined capacity $N(a')$ allows for a smoother approximation while remaining entirely discrete.

2.5 Example: Practical Application

In practice, these ideas are implemented in scripts such as `grains_arithmetic.py` and `PDE_3.py`. These scripts demonstrate:

- **Basic Arithmetic:** Performing addition, subtraction, multiplication, and division on grains-coded fractions without using floating-point approximations.
- **Geometric Applications:** Approximating the perimeter of a circle using a regular polygon approach. The side lengths are computed using grains-coded steps, and their sum yields an exact perimeter.
- **No π Required:** Even though classical circle geometry involves π and trigonometric functions, the grains-coded method approximates a circle solely through discrete operations.

Script Explanation:

1. **Grain Class:** A class (e.g., `Grain`) encapsulates a grains-coded fraction. It supports arithmetic operations using integer arithmetic. A helper method `to_float()` is provided solely for debugging.

2. **grains_polygon_perimeter(...):** This function computes a polygon's perimeter using grains-coded side lengths, ensuring every step is exact.

3. **grains_approx_circle_perimeter(...):** This function naively approximates a circle's perimeter by dividing the diameter into segments and summing their lengths. There is no direct use of π or trigonometric functions—only exact, discrete operations.

3 Additional Theoretical Foundations: Formal Proof Sketches

The following outlines why grains-coded operations preserve standard arithmetic properties:

Correctness of Basic Operations

- **Commutativity and Associativity:** For any grains-coded fractions $\frac{k}{N(a)}$ and $\frac{m}{N(a)}$, addition is defined as:
$$\frac{k}{N(a)} + \frac{m}{N(a)} = \frac{k+m}{N(a)}.$$
Since integer addition is commutative and associative, these properties carry over.

- **Multiplication:** Multiplication is defined as:
$$\frac{k}{N(a)} \times \frac{m}{N(a)} = \frac{k \cdot m}{N(a)^2}.$$
After unifying denominators via refinement, the underlying integer operations ensure that commutativity and associativity hold.

Distributivity

The distributive law:

$$\left(\frac{k}{N(a)} + \frac{m}{N(a)}\right) \times \frac{r}{N(a)} = \frac{k}{N(a)} \times \frac{r}{N(a)} + \frac{m}{N(a)} \times \frac{r}{N(a)},$$

holds because it directly mirrors the standard distributive property of integers. This guarantees that all grains-coded arithmetic operations are exact.

Completeness for Rational Operations

Any rational number $\frac{k}{d}$ can be represented by embedding it into a refined capacity $N(a')$ where $N(a')$ is a multiple of d. This demonstrates that our finite system is complete for rational arithmetic.

Avoidance of Real-Number Infinity

Every operation is carried out in a finite number of integer-based steps. There is no need for infinite series or limits, which ensures all computations remain both exact and computationally feasible.

Practical Impact

- **Robustness:** Mapping grains-coded operations directly to standard rational arithmetic eliminates floating-point errors.
- **Implementation Guidance:** When coding your grains class, incorporate methods for unifying denominators and performing gcd-based simplification to maintain arithmetic integrity.
- **Bounded Resources:** By imposing a maximum capacity, the system ensures that while any rational can be represented exactly, the operations remain within practical resource limits.

4 Conclusion

The grains-coded approach to numbers and arithmetic successfully replicates classical rational arithmetic using finite, discrete increments. Key benefits include:

- **Exactness:** Every arithmetic operation is performed exactly via integer-based steps.
- **Completeness:** Any rational number can be represented through capacity refinement.
- **Finite Nature:** There is no reliance on infinite sets or continuous approximations.
- **Resource Awareness:** A practical capacity bound ensures that all computations remain within real-world limits.

This method not only avoids floating-point rounding errors but also provides a robust foundation for computational systems that require precise, resource-aware arithmetic. As such, the grains-coded framework is both a theoretical breakthrough and a practical tool for modern IT applications.