Chapter 5: Algebraic Structures (Groups, Rings, Fields, Vector Spaces)

Probabilistic Minds Consortium (voids.blog)

2024/2025

1 Introduction

Most of us learn that solving equations, building transformations, and manipulating coordinates rely on well-known algebraic structures: groups, rings, fields, and vector spaces. In a finite, grains-coded system, these structures remain valid but depend entirely on discrete increments of the form $\frac{k}{N(a)}$. In this section, we illustrate how:

- Increments form additive groups (referred to as “VOID groups”).
- Multiplication extends these groups to field-like sets.
- Vector spaces are built over these grains-coded fields.
- Matrices and linear maps operate in a discrete manner using grains-coded addition and multiplication.

This section demonstrates that the fundamental building blocks of algebra naturally emerge from a system that is entirely finite and discrete. Instead of relying on the abstract continuum of numbers, we work with well-defined, integer-based fractions that mimic the behavior of classical rational numbers. This approach avoids the pitfalls of infinite sets and floating-point arithmetic while still enabling the full range of algebraic manipulations needed for solving equations and modeling transformations.

2 VOID Groups: The Additive Side

2.1 Systematic Construction

Step 1: Start with a Vantage a.
Recall from earlier sections that for each observer state a we have a capacity $V(a)$ and an integer function $N(a)$. This capacity sets our grains-coded increments:

$$\Big\{0,\ \frac{1}{N(a)},\ \frac{2}{N(a)},\ \ldots,\ \frac{N(a)}{N(a)} = 1\Big\},$$

but typically we also allow negative numerators for completeness.

Step 2: Extend Numerators to $\mathbb{Z}$.
To allow negative increments (necessary for subtraction and inverses), let k run through a finite integer range. For example, choose $-k_{\max} \le k \le k_{\max}$ with $k_{\max} = N(a)$. Then define:

$$\text{Increments}(a)\ =\ \Big\{\frac{k}{N(a)}\ \Big|\ k \in \mathbb{Z},\ -N(a) \le k \le N(a)\Big\}.$$

Step 3: Define Addition.
On these increments, define grains-coded addition by:

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

If $k+m$ falls outside the range $\{-N(a), \ldots, N(a)\}$, then either a unification of vantage (see Section 3) is required or the value must be embedded into a refined capacity a' with larger $N(a')$.

Step 4: Check Group Axioms.

- **Closure:** The sum $\frac{k+m}{N(a)}$ remains within the set if $-N(a) \le k+m \le N(a)$.
- **Identity:** $\frac{0}{N(a)}$ serves as the additive identity.
- **Inverse:** For each $\frac{k}{N(a)}$, the additive inverse is $\frac{-k}{N(a)}$.
- **Associativity:** Standard fraction addition is associative; therefore, grains-coded addition is as well.

Thus, the increments at a given vantage a form a finite additive group—commonly referred to as a VOID group within this grains-coded setting. This construction demonstrates that even with the simplest finite integers and a fixed denominator, all standard group operations can be performed exactly without rounding errors.

2.2 Basic Definition

Formally, a VOID group at vantage a is defined as:

$$\text{Increments}(a) \;=\; \Big\{ \frac{k}{N(a)} \;\Big|\; k \in \mathbb{Z},\; -N(a) \le k \le N(a) \Big\},$$

with addition given by

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

identity element $\frac{0}{N(a)}$, and inverse of $\frac{k}{N(a)}$ defined as $\frac{-k}{N(a)}$. Each observer state a thereby yields a finite set of grains-coded rationals. When more precision is needed, one refines capacity (see Section 3) or unifies vantage sets while preserving the group structure.

2.3 Example in Code

```
def grains_zero():
    return grain(0, 1)  % returns 0/1, the additive identity

def negate(x: grain) -> grain:
    return grain(-x.n, x.d)
```

In this example:

- `grains_zero()` returns $\frac{0}{1}$.
- `negate(x)` returns the additive inverse $\frac{-x.n}{x.d}$.

These functions illustrate the fundamental group properties in a grains-coded system.

3 Building a Field: Adding Multiplication

Extending from additive groups to field-like structures is a crucial step. Once multiplication is defined on the increments $\frac{k}{N(a)}$, the system emulates a field—a discrete subset of the rational numbers within each observer state.

1. **Rational Multiplication:**

$$\frac{k}{N(a)} \times \frac{m}{N(a)} = \frac{k \cdot m}{N(a) \cdot N(a)}.$$

At a single vantage, the multiplication is straightforward. For different denominators, unification or embedding is required.

2. **Multiplicative Identity:** $\frac{N(a)}{N(a)} = 1$ serves as the multiplicative identity.

3. **Inverses:** For any nonzero $\frac{k}{N(a)}$, the multiplicative inverse is given by $\frac{N(a)}{k}$ (provided that $|k| \leq N(a)$ so that the inverse remains within the finite set).

Code Example for Multiplication:

```
def grains_one():
    return grain(1, 1)  % Represents 1/1

class grain:
    def __init__(self, n, d=1):
        self.n = n
        self.d = d
        # Optionally, perform gcd reduction here

    def __mul__(self, other):
        new_num = self.n * other.n
        new_den = self.d * other.d
        return grain(new_num, new_den)

    def to_float(self):
        return self.n / self.d
```

This prototype implements multiplication on grains-coded fractions. In a complete system, you would also add division (with proper handling of zero), subtraction, and negation. The key is that all operations are performed using integer arithmetic, ensuring exactness without floating-point errors.

4 Vector Spaces Over Grains-Coded Fields

A grains-coded vector of dimension n at a given observer state a is defined as:

$$\Big(\frac{k_1}{N(a)}, \frac{k_2}{N(a)}, \ldots, \frac{k_n}{N(a)}\Big).$$

Vector addition and scalar multiplication are performed component-wise, using the grains-coded operations defined above.

Example:

```
v1 = [grain(2),      grain(5)]     % Represents (2/1, 5/1)
v2 = [grain(3,2),    grain(-1,1)]   % Represents (3/2, -1/1)

% Vector addition:
v_sum = [v1[i] + v2[i] for i in range(len(v1))]

% Scalar multiplication (multiplying by 2/3):
scalar = grain(2,3)
v_scaled = [scalar * comp for comp in v1]
```

This construction mirrors classical vector space theory but with every number represented as a finite, discrete fraction.

5 Matrices and Linear Maps

Matrices in a grains-coded system are two-dimensional arrays whose entries are grains-coded fractions. Standard operations—such as addition, multiplication, and inversion—are defined in this discrete framework.

- **Matrix Addition:** Performed entry-wise using grains-coded addition.
- **Matrix Multiplication:** The product is computed as the sum of grains-coded products of corresponding entries. Unification of denominators may be necessary for consistent arithmetic.
- **Row Operations and Inversion:** Techniques such as Gauss–Jordan elimination can be adapted to work entirely with grains-coded arithmetic.

Example Gauss-Jordan:

```
def gauss_jordan_solve(A, b):
    """
    Given a grains-coded NxN matrix A and a grains-coded vector b,
    perform row operations to obtain the solution vector x.
    (Assume necessary unifications and inverses are implemented.)
    """
    % (Pivot selection, row reduction, inverse calculation, etc.)
    ...
    return x
```

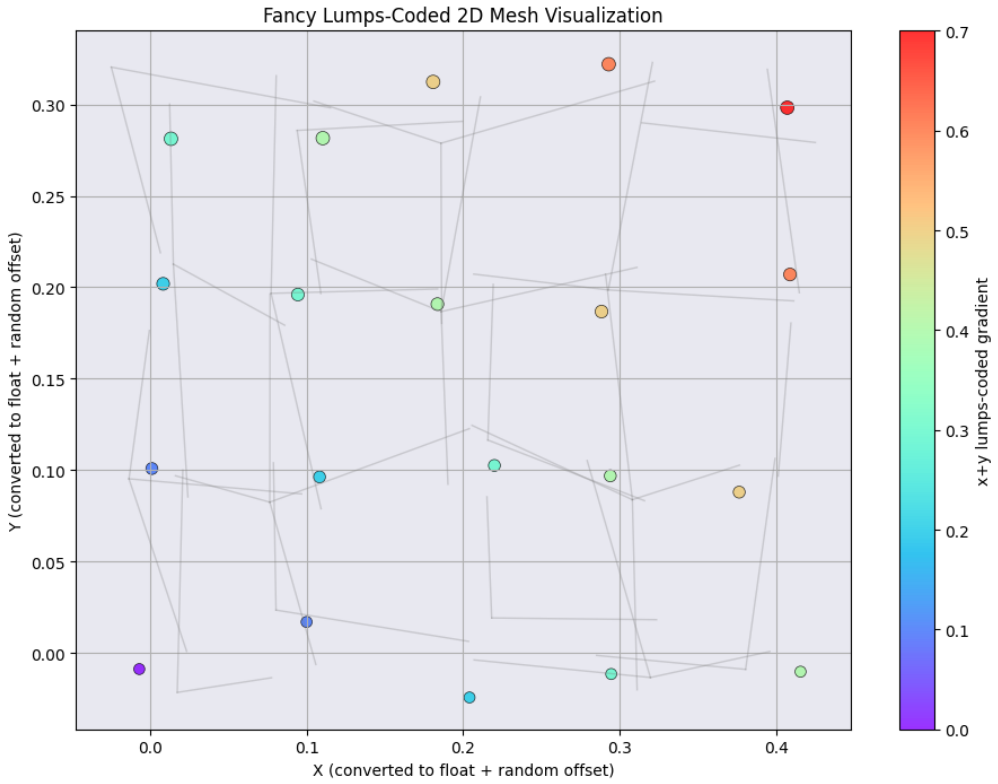
This function would illustrate that even complex linear algebra operations are performed exactly using our grains-coded approach.

6 A Mini Domain-Specific Language (DSL) for Grains-Coded Algebra

To enforce grains-coded rules and avoid any accidental use of floating-point arithmetic, you can develop a small DSL with custom classes. For instance:

```
class grain:
    def __init__(self, n, d=1):
        self.n = n
        self.d = d  # Optionally, simplify the fraction

    def __add__(self, other):
        % Unify denominators (or assume common denominators)
        return grain(self.n * other.d + other.n * self.d, self.d * other.d)

    def __mul__(self, other):
        return grain(self.n * other.n, self.d * other.d)

    def to_float(self):
        return self.n / self.d

    % Implement __sub__, __truediv__, __neg__, etc.
```

Additional classes for vectors and matrices would operate on lists of `grain` objects, ensuring that every arithmetic operation is performed in a precise, grains-coded manner.



7 Putting It All Together

The systematic construction of algebraic structures in our finite system proceeds from basic grains-coded increments (VOID groups) through the addition of multiplication to form a field-like structure. Over this field, vector spaces are built and then extended to matrices and linear maps. The key points are:

- **VOID Groups:** Grains-coded increments (allowing negative numerators) form finite additive groups at each observer state. No hidden continuity or infinite expansions exist.
- **Field-Like Sets:** Once multiplication is defined, the set of grains-coded fractions behaves like a field, where every nonzero element has a multiplicative inverse (within the capacity constraints).
- **Vector Spaces:** Coordinate vectors and matrices can be built over these fields, enabling all standard linear algebra operations.

- **Exactness:** Every operation is performed using exact, integer arithmetic, ensuring no rounding errors or floating-point approximations.

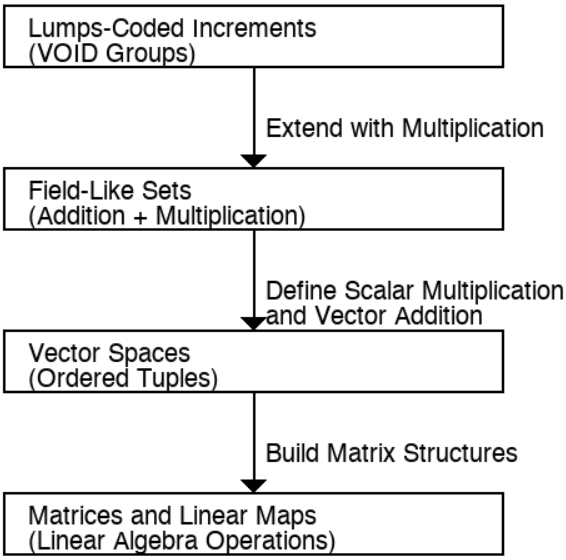
Main Advantage: We replicate the full scope of classical algebraic operations without invoking the continuum. Every arithmetic step is finite, discrete, and grains-coded—aligning with our capacity-based approach that is both mathematically rigorous and fully implementable.

8 Final Remarks: Building from the Ground Up

- **Stepwise Construction:** We start with an observer state a and its finite set of increments, which naturally forms an additive (VOID) group. Extending these operations with multiplication produces a field-like structure. From there, we construct vector spaces and matrices. Each step follows logically from the previous one.

- **Practical Use:** In practice, you will implement grains-coded classes, ensure proper denominator unification, and expand capacity as needed. Scripts such as $\texttt{grains}_a rithmetic.pyand$

The construction presented in this section shows that the entire edifice of algebra—groups, rings, fields, vector spaces, and matrices—can be rebuilt from the ground up using finite, grains-coded increments. This approach not only avoids floating-point errors but also provides a robust foundation for advanced computational systems.



`