A Rigorous Triadic Framework for Neurosymbolic Reasoning

José Arturo Ornelas Brand
Independent Researcher
arturoornelas62@gmail.com

November 14, 2025

Abstract

Current Large Language Models (LLMs) excel at statistical pattern matching but struggle with verifiable symbolic logic. This paper proposes a formal, integer-based relational framework as a candidate model for the underlying logic of emergent "sparse circuits" [1]. The framework is built on **dual functions**: a *generative* function (Φ_G) for predicting new relations and a *discovery* function (Φ_D) for inferring balancing rules from existing data. It uses integer balancing and normalization via the Greatest Common Divisor (GCD) to compute relational transformations, moving beyond floating-point vector addition to a **ratio-based**, symbolic logic. We first define the formal mathematics. We then validate its descriptive power by modeling the laws of classical mechanics. Finally, we discuss its implementation in Python using exact rational arithmetic (`fractions`) and graph libraries (`networkx`) for building hybrid neurosymbolic architectures. We provide a Python implementation to validate the framework's computational viability.

Keywords: Neurosymbolic AI, Relational Framework, Interpretability, Sparse Circuits, Knowledge Graphs, Greatest Common Divisor (GCD), Symbolic Reasoning, Rational Arithmetic, Automated Scientific Discovery, Hybrid Neurosymbolic Systems, Mechanistic Interpretability

Contents

1 Introduction: The Need for a Symbolic Logic in AI

The field of Artificial Intelligence has been revolutionized by Large Language Models (LLMs). Their ability to perform analogical reasoning, such as the famous $\vec{\text{King}} - \vec{\text{Man}} + \vec{\text{Woman}} \approx \vec{\text{Queen}}$ operation [3], is a product of high-dimensional vector arithmetic in a continuous, floating-point space. While powerful, this *additive* approach lacks the rigorous, verifiable, and discrete logic required for many scientific domains. Recent research on "sparse circuits" [1] suggests that LLMs implicitly learn functional subgraphs. However, a formal language to describe the *logic* of these circuits is missing. We are left with statistical correlations rather than explainable computations. This paper proposes a candidate for this formal language: a **Triadic Relational Framework** based on normalized integers. We move from an additive relationship ($A - B + C = D$) to a **ratio-based** one ($\Phi(A, B, C) \to D$) grounded in integer balancing and the Greatest Common Divisor (GCD). We will first present the formal mathematics. We will then demonstrate its applicability with two case studies:

1. **Classical Mechanics:** Modeling the deterministic, formula-based laws of physics as a hierarchical graph.
2. **AI Interpretability:** Proposing this framework as a formal model for sparse circuits and neurosymbolic systems, including a novel semantic mapping scheme (as detailed in Sec. 4.4).

2 A Formal Triadic Relational Framework

This model is generalized into a formal mathematical framework which uses integer-based normalization to describe relational triads, designed to unify physical and abstract concepts.

2.1 The Dual Functions of the Framework

The framework is built upon two complementary functions: a Generative function (Φ_G) for prediction, and a Discovery function (Φ_D) for inference.

2.1.1 Generative Function (Φ_G): The 4-Step Process

The Φ_G function predicts a new concept C_4 from three inputs (C_1, C_2, C_3) and a known rule (a, b). This is a 4-step process:

1. **Input Concepts:** We start with three known concepts (inputs): C_1, C_2, C_3.
2. **Normalization:** We compute a common divisor for the inputs: $\text{GCD}_{\text{in}} = \text{GCD}(C_1, C_2, C_3)$. We then normalize the inputs to their integer base: $C'_i = C_i / \text{GCD}_{\text{in}}$.
3. **Relational Transformation:** A triadic relation Φ_G is applied. This relation is defined by a pair of minimal, co-prime integers ($a, b \in \mathbb{Z}^+$) that represent the "rule" of the triad. The transformation predicts a normalized output, C'_4.

   $$\Phi_G(C'_1, C'_2, C'_3) \to C'_4 \quad \text{where} \quad C'_4 = \frac{a \cdot C'_2 \cdot C'_3}{b \cdot C'_1}$$

   This is derived from the balancing equation: $a \cdot C'_2 \cdot C'_3 = b \cdot C'_1 \cdot C'_4$. Other Φ functions are also possible (see Sec 4.3).
4. **Denormalization:** The final, concrete output C_4 is retrieved by scaling the normalized output by the input divisor: $C_4 = C'_4 \cdot \text{GCD}_{\text{in}}$. The computation must result in an integer C_4, or the relation is considered invalid for this rule.

2.1.2 Discovery Function (Φ_D): Inferring Rules

Alongside the generative function, the framework includes a crucial **Discovery Function (Φ_D)**. This function operates in reverse: given a known, balanced set of four concepts (C_1, C_2, C_3, C_4), it infers the minimal, co-prime balancing coefficients (a, b) and the associated simplicity K.

$$\Phi_D(C_1, C_2, C_3, C_4) \to (a, b, K)$$

This is achieved by first normalizing all four inputs by their GCD ($\text{GCD}_{\text{in}} = \text{GCD}(C_1, C_2, C_3, C_4)$), and then computing the exact rational ratio $\frac{a}{b}$, which is simplified to its minimal co-prime form:

$$\frac{a}{b} = \frac{C'_1 \cdot C'_4}{C'_2 \cdot C'_3}$$

The simplicity is then $K = 1/(a \cdot b)$. This function is essential for analyzing existing data and discovering the underlying rules of a system (see Sec 3.2). Note that the assignment of roles (e.g., which concept is C_1 vs. C_2) is a crucial modeling step, as it determines the sides of the balance equation. This assignment is typically performed by the researcher or, in a neurosymbolic system, learned by the neural component (see Sec 4.4).

2.2 Key Components of the Framework

The framework separates the logic into node-level computations and network-level connections for clarity:

- **Node Logic (Triad Δ):** The simplest stable unit is a triad $\Delta(C_1, C_2, C_3; K)$, where the relational mechanism Φ (e.g., Φ_G or Φ_D) computes the output C'_4 from normalized inputs C'_i using co-prime a, b. The triad is quantified by a simplicity constant $K = 1/(a \cdot b)$. This metric measures the simplicity of the rule: higher K indicates simpler rules (e.g., $a = 1, b = 1$ yields $K = 1$, the maximum simplicity). Normalized C'_i bridge physical and abstract domains.
- **Network Logic ($\mathcal{N}$):** Triads connect to form graphs. The output C_4 of one triad seeds new triads, modeling dynamic systems. Connectivity is defined by $\mathcal{N}(\Delta_i, \Delta_j) = w_{ij} \cdot A_{ij}$, where A_{ij} is the adjacency matrix (1 if C_4 from Δ_i inputs to Δ_j, 0 otherwise), and $w_{ij} = K_i$ (the simplicity of Δ_i) prioritizes paths with fundamental rules.

2.3 Chaining Triads and Networks

The network logic enables chaining: The output of one triad, C_4, can serve as an input for a new triad, $\Delta_2(C_4, C_5, C_6) \to C_7$. This forms a chain or graph. This is implemented as a **directed graph** (using `networkx`), where each node Δ_i stores its simplicity K_i. For example, if $\Delta_1(18, 6, 8) \to C_4 = 2$ (from Sec. 2.4), its output $C_4 = 2$ can feed Δ_2:

- Let Δ_2 be $\Delta_2(C_4 = 2, C_5 = 5, C_6 = 10) \to C_7$.
- Assume Δ_2 has a simple rule $a = 1, b = 1$ ($K_2 = 1$).
- $\text{GCD}_{\text{in}} = \text{GCD}(2, 5, 10) = 1$. So $C'_i = C_i$.
- $C'_7 = \Phi_G(2, 5, 10) = \frac{1 \cdot 5 \cdot 10}{1 \cdot 2} = 25$.
- $C_7 = C'_7 \cdot 1 = 25$.
- This creates a chain $\Delta_1 \to \Delta_2$ producing a final output of 25. The edge $\Delta_1 \to \Delta_2$ would have weight $w_{12} = K_1 = 1/12$.

2.4 Abstract Numerical Example (Using Φ_G)

Consider an abstract system with three inputs: $C_1 = 18$, $C_2 = 6$, $C_3 = 8$. We wish to apply a rule defined by $a = 3, b = 4$.

1. **Input Concepts**: $C_1 = 18$, $C_2 = 6$, $C_3 = 8$.
2. **Normalize**: $\text{GCD}_{\text{in}} = \text{GCD}(18, 6, 8) = 2$.
   - $C'_1 = 18/2 = 9$
   - $C'_2 = 6/2 = 3$
   - $C'_3 = 8/2 = 4$
3. **Apply** Φ_G: We apply the rule $a = 3, b = 4$.
$$C'_4 = \frac{a \cdot C'_2 \cdot C'_3}{b \cdot C'_1} = \frac{3 \cdot 3 \cdot 4}{4 \cdot 9} = \frac{36}{36} = 1$$
4. **Denormalize (Revert)**:
$$C_4 = C'_4 \cdot \text{GCD}_{\text{in}} = 1 \cdot 2 = 2$$

Thus, the triad $(18, 6, 8)$ transforms into 2. The simplicity of this rule is $K = 1/(3 \cdot 4) = 1/12$.

3 Case Study 1: Modeling Classical Mechanics

To demonstrate the framework's descriptive power, we apply it to the concepts of classical mechanics.

3.1 Core Concepts as a K_3 Graph

We define three pillars of classical mechanics as a foundational graph: **Newton's Laws of Motion**, **Kinetic/Potential Energy**, and **Gravity/Acceleration**. In graph theory, this is a complete graph K_3.

- **Vertex A:** Newton's Laws of Motion
- **Vertex B:** Kinetic and Potential Energy
- **Vertex C:** Gravity and Acceleration

The three edges illustrate the interrelationships:

1. **Edge A$\leftrightarrow$B (Laws $\leftrightarrow$ Energy):** The work-energy theorem.
2. **Edge B$\leftrightarrow$C (Energy $\leftrightarrow$ Gravity):** Gravity (acceleration g) defines potential energy ($PE = mgh$).
3. **Edge C$\leftrightarrow$A (Gravity $\leftrightarrow$ Laws):** Gravity is a force, described by Newton's 2nd Law ($F_g = mg$).

3.2 Handling Physical Formulas and Coefficients

The framework's balancing coefficients (a, b) are ideal for handling physical formulas. Instead of manually mapping them, we use the **Discovery Function (Φ_D)** defined in Sec 2.1.2. For example, to analyze Kinetic Energy ($KE = \frac{1}{2}mv^2$, which is rewritten as $2 \cdot KE = 1 \cdot m \cdot v^2$), we analyze a known balanced set of concepts. Our framework's balance equation is $bC'_1C'_4 = aC'_2C'_3$. To map the physics equation, we assign roles as follows: $C_1 = KE$, $C_2 = m$, $C_3 = v^2$, and

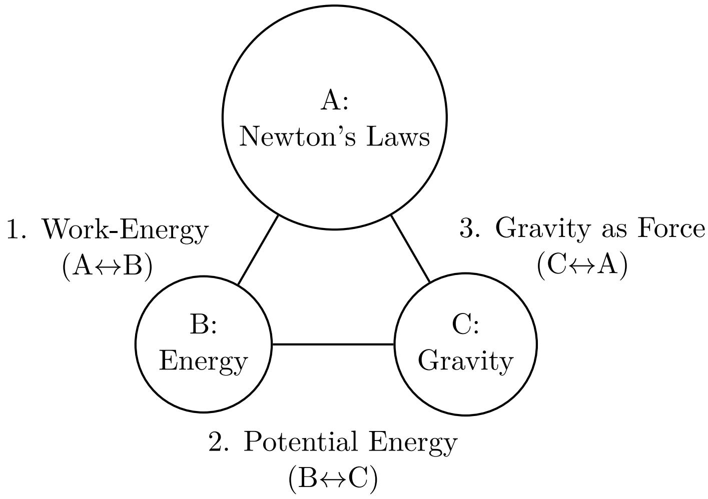

Figure 1: A K_3 complete graph showing the interrelationship between the three pillars of classical mechanics.

introduce a unit concept $C_4 = 1$ as a placeholder to complete the triad. This assignment implies $b \cdot KE \cdot 1 = a \cdot m \cdot v^2$, and we expect Φ_D to discover $a = 1$ and $b = 2$. Note that this role assignment is a crucial modeling step, determined by the structure of the formula (e.g., isolating the "output" or balancing sides). We test this with integer values that satisfy the relation, **matching those in our Appendix test code**: $KE = 1, m = 1, v = \sqrt{2} \to v^2 = 2$:

- We apply $\Phi_D(C_1 = 1, C_2 = 1, C_3 = 2, C_4 = 1)$.
- $\text{GCD}_{\text{in}} = \text{GCD}(1, 1, 2, 1) = 1$. All $C_i' = C_i$.
- The function computes the ratio based on the formal definition in Sec 2.1.2:

$$\frac{a}{b} = \frac{C_1' \cdot C_4'}{C_2' \cdot C_3'} = \frac{1 \cdot 1}{1 \cdot 2} = \frac{1}{2}$$

- It thus *discovers* the minimal balancing coefficients $a = 1, b = 2$, correctly matching the $2 \cdot KE = 1 \cdot m \cdot v^2$ formulation.

For formulas with irrational constants (e.g., $C = 2\pi r$), the framework can operate on integer-scaled approximations. We can approximate $\pi \approx 22/7$, leading to $7C \approx 44r$. We map this as $bC = ar$ (using $C_1 = C, C_2 = r, C_3 = 1, C_4 = 1$). Using Φ_D on known values $C_1 = 44, C_2 = 7, C_3 = 1, C_4 = 1$, it discovers the ratio:

$$\frac{a}{b} = \frac{C_1' \cdot C_4'}{C_2' \cdot C_3'} = \frac{44 \cdot 1}{7 \cdot 1} = \frac{44}{7}$$

This yields $a = 44, b = 7$, successfully discovering the integer-approximated rule. As a simpler example, Ohm's Law ($V = IR$) maps to a direct $a = 1, b = 1$ rule. By assigning roles $C_1 = V, C_2 = I, C_3 = R, C_4 = 1$ and testing known integer values (e.g., $V = 10, I = 2, R = 5$), Φ_D correctly discovers the simplest ratio $\frac{a}{b} = \frac{10 \cdot 1}{2 \cdot 5} = \frac{1}{1}$, yielding $a = 1, b = 1$ and the maximum simplicity $K = 1$.

4 Case Study 2: A Model for AI Interpretability

The triadic framework, demonstrated with physics, proposes a general-purpose model for relational logic. This formalism has profound implications for Artificial Intelligence, particularly in the growing field of neurosymbolic systems [4].

4.1 Formalizing Sparse Circuits

A primary challenge in AI interpretability is understanding the *logic* of emergent "sparse circuits" [1]. The framework proposed here offers a candidate model for formally describing this logic.

- Our model provides a path to move from a **statistical correlation** (the LLM's "black box") to an **explicit logical computation** (the Φ function with GCD normalization).

In essence, while LLMs find *what* concepts are related, our model offers a language to describe *how* and *why* they are related. This discrete approach could also reduce hallucinations in LLMs by enforcing verifiable, logical balances.

4.2 Integer-Based Relations vs. Vector Additivity

As noted, LLMs use analogical reasoning via vector arithmetic. This is an **additive** relationship. Our model proposes a **ratio-based** relationship defined by integers (a, b). This approach may offer a more stable and generalizable method for capturing symbolic analogies.

4.3 Formalizing the "King-Queen" Analogy

Let's contrast the two models for the famous analogy.

- **Vector (Additive) Model:**

$$\vec{\text{King}} - \vec{\text{Man}} + \vec{\text{Woman}} \approx \vec{\text{Queen}}$$

  This is a statistical approximation in a continuous vector space.

- **Triadic (Ratio-Based) Model:** The standard "A is to B as C is to D" analogy ($C_1 : C_2 :: C_3 : C_4$) is a ratio-based operation: $\frac{C_1}{C_2} = \frac{C_3}{C_4}$. This implies a generative function Φ_A (for Analogy):

$$\Phi_A(C_1, C_2, C_3) \to C_4 \quad \text{where} \quad C_4 = \frac{C_1 \cdot C_3}{C_2}$$

  This is a variant of our framework's Φ_G function (one where the inputs are arranged differently and $a = 1, b = 1$). Let's test this symbolic, integer-based math.

4.4 Mapping Semantic Concepts to Integers

A critical challenge is mapping abstract concepts like "King" to the integers C_i required by our framework. This mapping is the key "neural" component of a neurosymbolic system, where a model learns a function $f(\text{concept}) \to \mathbb{Z}$ based on latent factors (e.g., prime decompositions for orthogonal attributes). The success of our framework hinges on this learned representation: the neural layer must produce integers that enable balanced, simple triads (integer C_4 with high K).

To demonstrate feasibility, consider a simple scheme where attributes are mapped to primes (assuming orthogonality; in practice, a neural layer would learn these):

- $\text{Attribute(Royalty)} = 7$
- $\text{Attribute(Male)} = 3$
- $\text{Attribute(Female)} = 5$

Concepts are products:

- $C_1(\text{King}) = 7 \cdot 3 = 21$
- $C_2(\text{Man}) = 3$

- $C_3(\text{Woman}) = 5$

Applying Φ_A: $(\text{GCD}(21, 3, 5) = 1)$

$$C_4 = \frac{21 \cdot 5}{3} = 35 = 7 \cdot 5(\text{Queen})$$

This integer-based calculation succeeds. In a full system, the neural model could be trained with our framework as a loss function: minimize loss when predicted integers yield integer balances with high K.

5 Conclusion and Future Work

This paper has introduced a formal triadic relational framework based on integer balancing and GCD normalization. We have demonstrated its utility by modeling the deterministic laws of classical physics and by proposing it as a formal language for AI interpretability, built upon dual generative (Φ_G) and discovery (Φ_D) functions. This convergence of the neural and symbolic layers forms our primary path for future work. The initial goal of discovering new physical relationships from a complete "graph of physics," (as outlined in [2]) is computationally intractable to build manually. However, we propose using the **Neural Layer (LLM)** as an advanced mining tool to parse scientific literature and automatically extract the knowledge graph and candidate triads (C_1, C_2, C_3). More crucially, the LLM can learn the mapping $f(\text{concept}) \to \mathbb{Z}$, producing integer representations that enable balanced triads. The **Symbolic Layer (our Φ Framework)** will then analyze these triads. By using Φ_D to find triads with simple rules (high K value), our framework can act as a generative engine (using Φ_G) to predict new, verifiable formulas (C_4). This framework provides an ideal loss function for training the neural mapper: reward mappings that yield integer C_4 with high K, moving beyond simple analogy to true automated scientific discovery. **Limitations and Implementation:** A reference implementation of this framework has been developed in Python. It utilizes `math.gcd` for normalization, the `fractions` library to ensure exact rational arithmetic (eliminating floating-point errors in K), and `networkx` to construct and manage the relational knowledge graphs. This implementation confirms the computational viability of the dual Φ_G and Φ_D functions. Future work will focus on extending this deterministic, ratio-based model. This modularity can be achieved without altering the core logic. For instance, **probabilistic** relationships can be modeled by introducing distributions (e.g., via Monte Carlo sampling on a and b) to handle uncertainty. Similarly, **additive** relationships (common in LLM vector analogies) can be integrated by defining a hybrid Φ function (e.g., $\Phi_H = \Phi_G + \Phi_{Additive}$) that combines ratio-based and additive operations, bridging the gap to modern neural architectures. Further research will also investigate the framework's scalability, using efficient graph algorithms to manage networks of thousands of triads. We aim to implement a neurosymbolic prototype that utilizes this framework's balancing rules as a loss function for training on relational datasets, such as those found in standard analogy benchmarks.

Acknowledgments

AI tools were employed for auxiliary tasks such as assistance in the LaTeX formatting, Python code debugging, and iterative refinement of the prose in this manuscript. The core concepts, framework logic, and final conclusions remain the original work of the author.

References

[1] Marks, S., Rager, C., Michaud, E. J., Velikovi, P., Yacoby, Y., Kravec, S., Laidlaw, C., Aitchison,M., Prabakaran, S., McCaffrey, R., Graham, L., Duong, D. H., Perez, E., Bengio, Y., and Tegmark, M. (2024). Sparse Feature Circuits: Discovering and Editing Interpretable Causal Graphs in Language Models *arXiv preprint arXiv:2403.19647*.

[2] Romiti, M. (2025). A Graph-Based Framework for Exploring Mathematical Patterns in Physics: A Proof of Concept. *arXiv preprint arXiv:2508.05724*. https://arxiv.org/abs/2508.05724.

[3] Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. *arXiv preprint arXiv:1301.3781*.

[4] Garcez, A. d'A., & Lamb, L. C. (2023). Neurosymbolic AI: the 3rd wave. *Artificial Intelligence Review, 56(11):12387-12406*.

A Appendix: Core Python Implementation

This section contains the core Python code for the framework, demonstrating the implementation of the ‘TriadicRelationalFramework‘ (with its dual functions) and the ‘TriadicNetwork‘ class.

Listing 1: Core Python Implementation

```
import math
from fractions import Fraction
import networkx as nx

class TriadicRelationalFramework:
        def __init__(self):
                pass # No initial state needed for basic implementation

        def compute_triad(self, C1, C2, C3, a, b):
                """
                Compute the triadic relational transformation.

                Parameters:
                - C1, C2, C3: Input integer concepts
                - a, b: Positive integer balancing coefficients (minimal
    , co-prime)

                Returns:
                - C4: The computed output integer
                - K: The simplicity constant (Fraction: 1 / (a * b))
                - steps: Dictionary with intermediate steps for
    transparency
                """
                if not all(isinstance(x, int) and x > 0 for x in [C1, C2
    , C3, a, b]):
                        raise ValueError("All␣inputs␣must␣be␣positive␣
    integers.")

                steps = {}

                # Step 1: Input Concepts
                steps['inputs'] = {'C1': C1, 'C2': C2, 'C3': C3, 'a': a,
     'b': b}

                # Step 2: Normalization
                gcd_in = math.gcd(C1, C2, C3)
                C1_prime = C1 // gcd_in
                C2_prime = C2 // gcd_in
                C3_prime = C3 // gcd_in
                steps['normalization'] = {'gcd_in': gcd_in, 'C1_prime':
    C1_prime, 'C2_prime': C2_prime, 'C3_prime': C3_prime}

                # Step 3: Relational Transformation (Phi)
                numerator = a * C2_prime * C3_prime
                denominator = b * C1_prime
                C4_prime = Fraction(numerator, denominator)
                steps['transformation'] = {'C4_prime': str(C4_prime)}

                # Step 4: Denormalization
                C4 = C4_prime * gcd_in
                if C4.denominator != 1:
                        raise ValueError("The␣balancing␣does␣not␣result␣
```

```
in␣an␣integer␣C4_prime.␣Adjust␣a,␣b␣or␣inputs.")
            C4 = int(C4) # Convert to int if integer
            steps['denormalization'] = {'C4': C4}

            # Simplicity K
            K = Fraction(1, a * b)
            steps['K'] = str(K)

            return C4, K, steps

    def analogy_variant(self, C1, C2, C3):
            """
            Variant for analogies like King:Man :: Queen:Woman,
which is C4 = (C1 * C3) / C2.
            Assumes a=1, b=1, but adjusted order.

            Parameters:
            - C1: Starting concept (e.g., King)
            - C2: To remove (e.g., Man/Male)
            - C3: To add (e.g., Woman/Female)

            Returns:
            - C4: Predicted concept (e.g., Queen)
            - steps: Dictionary with intermediate steps
            """
            if not all(isinstance(x, int) and x > 0 for x in [C1, C2
, C3]):
                    raise ValueError("All␣inputs␣must␣be␣positive␣
integers.")

            steps = {}

            # Normalization (over inputs)
            gcd_in = math.gcd(C1, C2, C3)
            C1_prime = C1 // gcd_in
            C2_prime = C2 // gcd_in
            C3_prime = C3 // gcd_in
            steps['normalization'] = {'gcd_in': gcd_in, 'C1_prime':
C1_prime, 'C2_prime': C2_prime, 'C3_prime': C3_prime}

            # Analogy transformation: C4_prime = (C1_prime *
C3_prime) / C2_prime
            numerator = C1_prime * C3_prime
            C4_prime = Fraction(numerator, C2_prime)
            steps['transformation'] = {'C4_prime': str(C4_prime)}

            # Denormalization
            C4 = C4_prime * gcd_in
            if C4.denominator != 1:
                    raise ValueError("The␣analogy␣does␣not␣result␣in
␣an␣integer␣output.␣Check␣attribute␣mappings.")
            C4 = int(C4) # Convert to int if integer
            steps['denormalization'] = {'C4': C4}

            # For analogy, a=1, b=1 implicitly
            K = Fraction(1, 1) # 1.0 as Fraction
            steps['K'] = str(K)
```

```
        return C4, K, steps

    def check_static_balance(self, C1, C2, C3, C4):
        """
        Check static balance for existing formula (find minimal
co-prime a,b such that a C2' C3' = b C1' C4').

        Parameters:
        - C1, C2, C3, C4: Positive integers

        Returns:
        - a, b: Minimal co-prime balancing coefficients
        - K: Simplicity (1 / (a * b))
        - steps: Dictionary with steps
        """
        if not all(isinstance(x, int) and x > 0 for x in [C1, C2
, C3, C4]):
            raise ValueError("All␣inputs␣must␣be␣positive␣
integers.")

        steps = {'inputs': {'C1': C1, 'C2': C2, 'C3': C3, 'C4':
C4}}

        gcd_in = math.gcd(C1, C2, C3, C4)
        C1_prime = C1 // gcd_in
        C2_prime = C2 // gcd_in
        C3_prime = C3 // gcd_in
        C4_prime = C4 // gcd_in
        steps['normalization'] = {'gcd_in': gcd_in, 'C1_prime':
C1_prime, 'C2_prime': C2_prime,
                    'C3_prime': C3_prime, 'C4_prime': C4_prime}

        ratio = Fraction(C1_prime * C4_prime, C2_prime *
C3_prime)
        a = ratio.numerator
        b = ratio.denominator
        gcd_ab = math.gcd(a, b)
        a //= gcd_ab
        b //= gcd_ab
        steps['balancing'] = {'a': a, 'b': b}

        K = Fraction(1, a * b)
        steps['K'] = str(K)

        return a, b, K, steps

    def chain_triads(self, initial_C1, triad_list):
        """
        Chain multiple triads: Each triad is (C2, C3, a, b).
Output of one is C1 for next.

        Parameters:
        - initial_C1: Starting C1
        - triad_list: List of tuples [(C2, C3, a, b), ...]

        Returns:
        - final_C4: Final output
        - all_K: List of Ks
```

```
                - all_steps: List of steps dicts
                """
                current_C1 = initial_C1
                all_K = []
                all_steps = []
                for C2, C3, a, b in triad_list:
                        C4, K, steps = self.compute_triad(current_C1, C2
    , C3, a, b)
                        all_K.append(K)
                        all_steps.append(steps)
                        current_C1 = C4 # Chain
                return current_C1, all_K, all_steps

class TriadicNetwork:
        def __init__(self):
                self.graph = nx.DiGraph() # Directed for chaining

        def add_triad(self, triad_id, C1, C2, C3, a, b):
                framework = TriadicRelationalFramework()
                _, K, _ = framework.compute_triad(C1, C2, C3, a, b)
                self.graph.add_node(triad_id, K=K)

        def add_connection(self, from_id, to_id):
                if from_id in self.graph and to_id in self.graph:
                        w = self.graph.nodes[from_id]['K'] # Weight = K
    of from
                        self.graph.add_edge(from_id, to_id, weight=w)

        def visualize(self):
                # Manual print for compatibility with networkx >=3.0
                print(f"Graph with {self.graph.number_of_nodes()} nodes 
    and {self.graph.number_of_edges()} edges")
                for node, data in self.graph.nodes(data=True):
                        print(f"Node {node}: {data}")
                for from_node, to_node, data in self.graph.edges(data=
    True):
                        print(f"Edge {from_node} -> {to_node}, weight: {
    data['weight']}")

# Example Usage and Tests
framework = TriadicRelationalFramework()

# Test 1: Abstract Numerical Example from Paper
C4, K, steps = framework.compute_triad(18, 6, 8, 3, 4)
print("Abstract Example:")
print(f"C4: {C4}, K: {K}")
print("Steps:", steps)

# Test 2: King-Queen Analogy with Primes
C4_analogy, K_analogy, steps_analogy = framework.analogy_variant(21, 3,
    5)
print("\nKing-Queen Analogy:")
print(f"C4 (Queen): {C4_analogy}, K: {K_analogy}")
print("Steps:", steps_analogy)

# Test 3: Static Balance (e.g., 2 KE = m v^2, dummy values KE=1, m=1, v
    ^2=2, 'C4' as placeholder for balance)
a, b, K_static, steps_static = framework.check_static_balance(1, 1, 2,
```

```
    1) # Should give a=1, b=2
print("\nStatic␣Balance␣Example:")
print(f"a:␣{a},␣b:␣{b},␣K:␣{K_static}")
print("Steps:", steps_static)

# Test 4: Chaining (from paper example)
final_C, Ks, steps_list = framework.chain_triads(18, [(6, 8, 3, 4), (5,
    10, 1, 1)])
print("\nChaining␣Example:")
print(f"Final␣C:␣{final_C},␣Ks:␣{Ks}")
print("Steps␣List:", steps_list)

# Test 5: Network Example
net = TriadicNetwork()
net.add_triad('Delta1', 18, 6, 8, 3, 4)
net.add_triad('Delta2', 2, 5, 10, 1, 1)
net.add_connection('Delta1', 'Delta2')
print("\nNetwork␣Example:")
net.visualize()

# Fractional Example (e.g., approx pi in circumference C = 2pir approx
    2*(22/7)r)
# Predict r from C=44, dummy C2=1, C3=1, a=7, b=44 (inverted for demo;
    this will raise since not integer)
try:
        C4_frac, K_frac, steps_frac = framework.compute_triad(44, 1, 1,
    7, 44)
        print("\nFractional␣Example␣(pi␣approx):")
        print(f"C4␣(approx␣r):␣{C4_frac},␣K:␣{K_frac}")
except ValueError as e:
        print(f"\nFractional␣Example␣Error␣(expected␣for␣non-integer):␣{
    e}")
```
