NZM Braid Engine V0.3

This document records the *exact state* of the NZM braid substrate engine as developed and tested to date. It is intended as a **non-interpretive, non-governance, substrate-level record** for future contributors.

The scope is strictly limited to: - Braid word dynamics - Residue persistence (no-zero enforcement) - Admissible local moves - Structural (non-scalar) observables

No claims about physics, cryptography, governance, or applications are made here.

---

1. Conceptual Summary

1.1 Core Principles

The NZM braid engine implements a relational substrate in which:

- **Zero is unrepresentable by construction**
- **Cancellation produces structure, never erasure**
- **All evolution is local and topological**
- **Observables are descriptive, not scalar**

A braid word is treated as a *persistent relational state*, not a numeric value or algebraic object.

1.2 Residue Injection (Axiom VIII)

Classical annihilation:

```
σᵢ · σᵢ⁻¹ → ∅
```

NZM persistence rule:

```
[σᵢ, −σᵢ] → [1, 2, 1]
```

The motif `[1, 2, 1]` is treated as a **topological residue**: - It prevents collapse to the empty word - It encodes historical interaction - It recursively participates in future interactions

This ensures **ε > 0** at the substrate level.

1.3 Admissible Local Moves

Only the following moves are allowed:

1. **Adjacent Far Commutativity**
2. Only adjacent word positions may swap
3. Only if strand labels differ by ≥ 2
4. **Bidirectional Yang–Baxter Relation**

   $\sigma_i \sigma_{i+1} \sigma_i \leftrightarrow \sigma_{i+1} \sigma_i \sigma_{i+1}$

5. Implemented as a *pattern-level* rule
6. Orientation/chirality is not yet promoted to an observable

These moves define all allowed substrate evolution.

1.4 Structural Observables

The engine intentionally avoids numeric observables with ontological meaning.

Implemented observables are *descriptive signatures* only:

- **Persistence Signature**: `R{r}_C{c}`
- `r`: number of residue motifs `[1,2,1]`
- `c`: total braid word length
- **Stability Class**:
- `Stable`: unchanged under admissible normalization
- `PotentiallyReducible`: further evolution possible

These observables do not feed back into the dynamics.

1.5 Emergent Behavior (Observed, Not Assumed)

Repeated self-interaction experiments of the form:

```
A → A · A⁻¹ → normalize()
```

show that:

- Word length grows persistently
- Growth rate slows over time
- Residue count increases roughly linearly with length
- No collapse or runaway divergence occurs

When coarse-grained by external observers, this behavior is often summarized using a fitted fractal exponent:

```
D ≈ 1.161
```

This value is **not** present in the substrate or code and has no axiomatic role.

The **golden spiral** appears as a *local geometric motif* of residue turning, not as a global scaling law.

---

2. Reference Implementation

The following is the **exact code** used for all simulations and tests referenced above.

```
# topology.py - NZM-Substrate V0.3 (Production Build)
# Pure braid engine with no-zero persistence, admissible moves,
# and structural observables only.

from typing import List

RESIDUE_BRAID = [1, 2, 1]

class Braid:
    def __init__(self, generators: List[int]):
        if not generators:
            self.word = list(RESIDUE_BRAID)
        else:
            self.word = list(generators)
            self._reduce()

    def _reduce(self):
        stability_reached = False
        while not stability_reached:
            stability_reached = True
            i = 0
            while i < len(self.word) - 1:
                if self.word[i] == -self.word[i + 1]:
                    self.word[i:i + 2] = RESIDUE_BRAID
                    stability_reached = False
                    break
                i += 1
        if not self.word:
            self.word = list(RESIDUE_BRAID)

    def inverse(self) -> 'Braid':
        new_word = [-x for x in reversed(self.word)]
        return Braid(new_word)

    def __mul__(self, other: 'Braid') -> 'Braid':
        return Braid(self.word + other.word)

    def __repr__(self):
        return f"Braid({self.word})"

    def __eq__(self, other):
        if isinstance(other, Braid):
            return self.word == other.word
        return False

    def apply_far_commute(self, i: int, j: int) -> 'Braid':
        if not (0 <= i < len(self.word) and 0 <= j < len(self.word)):
            raise IndexError("Invalid positions")
        if abs(i - j) != 1:
            raise ValueError("Topological Violation: Can only commute adjacent word indices.")
        gen_i, gen_j = self.word[i], self.word[j]
        if abs(gen_i - gen_j) < 2:
            raise ValueError("Strand Entanglement: generators too close to commute.")
        new_word = list(self.word)
        new_word[i], new_word[j] = new_word[j], new_word[i]
        return Braid(new_word)

    def apply_yang_baxter(self, pos: int) -> 'Braid':
        if pos + 2 >= len(self.word):
            return self
        a, b, c = self.word[pos:pos + 3]
        new_word = list(self.word)
        if a == c and abs(a - b) == 1:
            new_word[pos:pos + 3] = [b, a, b]
            return Braid(new_word)
        elif b == c and abs(a - b) == 1:
            new_word[pos:pos + 3] = [a, c, a]
            return Braid(new_word)
        return self
    def shift_generator_left(self, pos: int, steps: int) -> 'Braid':
        current = self
        for _ in range(steps):
            if pos <= 0:
                break
            try:
                current = current.apply_far_commute(pos - 1, pos)
                pos -= 1
            except ValueError:
                break
        return current

    def normalize(self, max_iterations: int = 100) -> 'Braid':
        current = self
        iteration = 0
        while iteration < max_iterations:
            changed = False
            for pos in range(len(current.word) - 2):
                new = current.apply_yang_baxter(pos)
                if new != current:
                    current = new
                    changed = True
            i = 0
            while i < len(current.word) - 1:
                try:
                    new = current.apply_far_commute(i, i + 1)
                    if new != current:
                        current = new
                        changed = True
                except ValueError:
                    pass
                i += 1
            if not changed:
                break
            iteration += 1
        return current

    def persistence_signature(self) -> str:
        r = sum(1 for i in range(len(self.word) - 2)
                if self.word[i:i + 3] == RESIDUE_BRAID)
        c = len(self.word)
        return f"R{r}_C{c}"

    def stability_class(self) -> str:
        normalized = self.normalize()
        return "Stable" if normalized.word == self.word else
"PotentiallyReducible"
```

3. Status

This document represents a **stable checkpoint** for NZM braid development.

Any future changes should explicitly note: - Which axioms are being extended or modified - Whether new observables are descriptive or ontological - Whether new moves preserve no-zero persistence

Until then, this file should be treated as the authoritative reference for NZM Braid Engine V0.3.