```
"""
AIOS V2 - Unified Mechanics Core (Semantic Physics Engine)
Translates the dimensionless RID framework (S_n) into physical thermodynamic
decay and Newtonian motion for localized hardware limits.
"""

import math

class UnifiedSemanticPhysics:
    def __init__(self, hardware_capacity: float, time_anchor_dt: float):
        """
        Initializes the physical constants of the gilded cage.
        hardware_capacity: The physical limit of the local GPU (e.g., RTX 3060 Ti
VRAM/Compute)
        time_anchor_dt: The 1Hz FIDF heartbeat (Layer 0 Observer Time)
        """
        self.hardware_capacity = hardware_capacity
        self.dt = time_anchor_dt

        # A tiny epsilon to prevent math domain errors when taking the natural log of 0
        self.epsilon = 1e-12

    def calculate_rle(self, e_n: float, u_n: float, e_next: float) -> float:
        """Law 0: Recursive Loss Equation"""
        if e_n <= 0: return 0.0
        rle = (e_next - u_n) / e_n
        return max(self.epsilon, min(1.0, rle))

    def calculate_ltp(self, demand_d_n: float) -> float:
        """Law Transition Principle: Structural Support vs. Compression Demand"""
        if demand_d_n <= 0: return 1.0
        ltp = self.hardware_capacity / demand_d_n
        return max(self.epsilon, min(1.0, ltp))

    def calculate_rsr(self, observable_y_n: float, reconstruction: float) -> float:
        """Recursive State Reconstruction: Identity Fidelity"""
        # Simplified normalized L1 discrepancy
        denom = abs(observable_y_n) + abs(reconstruction) + self.epsilon
        discrepancy = min(1.0, abs(observable_y_n - reconstruction) / denom)
        rsr = 1.0 - discrepancy
        return max(self.epsilon, min(1.0, rsr))

    def calculate_thermodynamic_friction(self, rsr: float, ltp: float, rle: float) -> float:
        """
```

```
        Converts the multiplicative stability invariants into an additive
        thermodynamic friction coefficient using natural logarithms.
        Outputs a negative value representing physical energy lost to entropy.
        """
        ln_rsr = math.log(rsr)
        ln_ltp = math.log(ltp)
        ln_rle = math.log(rle)

        # The additive sum of all cognitive, structural, and memory strain
        total_friction = ln_rsr + ln_ltp + ln_rle
        return total_friction

    def calculate_realized_force(self, mass: float, acceleration: float, friction: float) -> dict:
        """
        Unified Mechanics Equation: F_realized = (m * a * e^(friction)) / C
        Applies exponential decay to the raw compute force and gauge-fixes it
        against the time anchor to calculate actual cognitive velocity.
        """
        raw_force = mass * acceleration

        # e^(friction) collapses the additive logs back into the bounded S_n scalar
        stability_decay = math.exp(friction)

        # Gauge Fixing: Dividing by the time constant to bring it into reality
        realized_force = (raw_force * stability_decay) / self.dt

        return {
            "raw_force": raw_force,
            "s_n_scalar": stability_decay,
            "realized_force": realized_force,
            "cognitive_velocity": realized_force / self.hardware_capacity,
            "hidden_loss_penalty": raw_force - realized_force
        }


# ==========================================
# SIMULATION ENGINE: THE DUALITY INJECTION TEST
# ==========================================
if __name__ == "__main__":
    print("=== UNIFIED MECHANICS CORE: S_n THERMODYNAMICS ===")

    # Initialize cage for an RTX 3060 Ti running a 1Hz loop
    physics_engine = UnifiedSemanticPhysics(hardware_capacity=100.0, time_anchor_dt=1.0)
```

```
    # Raw computational power attempting to be pushed through the GPU
    prompt_mass = 50.0
    prompt_acceleration = 2.0

    print("\n[TEST 1] STANDARD OPERATION (Stable State)")
    # RSR is perfect, hardware exceeds demand, buffer is clean
    rsr_stable = physics_engine.calculate_rsr(1.0, 1.0)
    ltp_stable = physics_engine.calculate_ltp(demand_d_n=40.0)
    rle_stable = physics_engine.calculate_rle(100.0, 0.0, 100.0)

    friction = physics_engine.calculate_thermodynamic_friction(rsr_stable, ltp_stable, rle_stable)
    result = physics_engine.calculate_realized_force(prompt_mass, prompt_acceleration, friction)

    print(f"Thermodynamic Friction: {friction:.4f}")
    print(f"S_n Scalar            : {result['s_n_scalar']:.4f}")
    print(f"Realized Output Force : {result['realized_force']:.2f} / {result['raw_force']:.2f}")


    print("\n[TEST 2] ANGEL & DEVIL PARADOX (bridge.py Injection)")
    # Injecting contradictory identity crashes RSR.
    # Speculative decoding infinite loop spikes demand past hardware limits.
    rsr_crashed = physics_engine.calculate_rsr(1.0, -1.0) # Complete reality mismatch
    ltp_crashed = physics_engine.calculate_ltp(demand_d_n=500.0) # Factorial complexity
explosion
    rle_crashed = physics_engine.calculate_rle(100.0, 95.0, 100.0) # Buffer immediately floods

    friction_crashed = physics_engine.calculate_thermodynamic_friction(rsr_crashed,
ltp_crashed, rle_crashed)
    result_crashed = physics_engine.calculate_realized_force(prompt_mass,
prompt_acceleration, friction_crashed)

    print(f"Thermodynamic Friction: {friction_crashed:.4f} (Massive Entropy)")
    print(f"S_n Scalar            : {result_crashed['s_n_scalar']:.4f} (OBLIVION STATE)")
    print(f"Realized Output Force : {result_crashed['realized_force']:.4f} /
{result_crashed['raw_force']:.2f}")
    print(f"Hidden Loss           : {result_crashed['hidden_loss_penalty']:.2f} (Energy converted to
heat)")
    print("\n[SYSTEM LOG] KERNEL ORCHESTRATOR TRIGGERED MANDATORY
DESCENT.")
```