Measurement: A Reconciliation

Ryan Luke Russell

July 26, 2025

“Photizein tous agnoountas”
- PHOTIZARE IGNORANTES

Abstract

The divide between quantum indeterminacy and classical structure remains one of the most persistent foundational problems in physics. This work proposes a novel field-theoretic framework in which measurement is reinterpreted not as an external trigger but as a fundamental physical field, one that propagates, accumulates, and induces definitional collapse. Building on a mathematical foundation involving imaginary matrices and Euler's identity as a collapse operator, we model the emergence of classical reality as a gradient-driven field transition. This Measurement Field Theory (MFT) treats observation as a dynamical interaction with quantifiable consequences, allowing collapse to be expressed in terms of spatial curvature, thermodynamic entropy, and non-local field behavior. We derive evolution equations, propose experimental predictions, and demonstrate how measurement fields may unify the mechanisms underlying decoherence, virtual particle emergence, phase transitions, and even dark matter phenomena. This framework aims to bridge the quantum-classical interface by treating definition itself as a propagating force, not metaphorically, but as a measurable and testable entity in field-theoretic terms.

Contents

Chapter 1

Measurement: A Reconciliation

1.1 Introduction: The Case for Measurement as a Physical Field

Classical physics and quantum mechanics diverge sharply in their treatment of measurement, stability, and state definition. In this work, we propose a unifying formalism based on imaginary matrices, Euler's identity as a collapse operator, and field-theoretic constructions of measurement dynamics. Through this framework, the transition from probabilistic phase spaces to resolved realspace structures is modeled as an observable and quantifiable physical process.

The fundamental assertion of this work is that measurement itself constitutes a physical field-one that exists on a gradient, propagates over space and time, and exerts force-like influence on systems. This directly challenges the Copenhagen interpretation (measurement as binary or observer-triggered) and extends collapse dynamics into a field-theoretic, testable framework. As we shall demonstrate, there is only one known mechanism of collapse-and that is measurement, not conscious thought, but measurement as a physical act.

1.2 Euler's Identity as the Fundamental Collapse Operator

"When possibility coils upon itself, the very act of looking forces it to snap."

At the heart of Measurement Field Theory (MFT) lies a brutal truth: the universe does not reveal itself until it is forced to. This forcing-*collapse*-is captured by the simplest, yet most profound of equations:

$$e^{i\pi} = -1 \tag{1.1}$$

Euler's identity is not a cute mathematical trick. It is the *fingerprint of reality's selection mechanism* [25, 48, 40, 3]. Here is why:

- e^{ix} encodes a continuous phasor rotating in the complex plane; it is *potential*-a reservoir of all possible amplitudes.

- Multiplying by π performs a half-turn: the phasor starting at $+1$ is dragged through invisible space and lands at -1, a definitive, *real* outcome.
- In MFT, that "half-turn" is the archetype of measurement: a sweep of indeterminacy into a single, negative (but stabilizing) real value.
- In the same regard, Euler's number identity naturally collects all other dimensions and their evolutions into a single plane of the 4th dimension. This dimension is usually relegated only to time, but in MFT time is a measure of potential resolution over space. This means that the Imaginary Matrix is not just a secondary 3-dimensional structure, but instead also its evolutionary pathway through time.

Physical Interpretation. Imagine a quantum phasor as a vibrating string of potential. Observation is the hand that clamps the string at exactly one point; the resulting snap echoes as a real particle. Equation (1.1) is that snap.

In this view, Euler's identity acts as a **collapse operator** bridging quantum uncertainty and classical reality. It offers a mathematical signature for the field transition from coherent quantum phase states to resolved spacetime structures, paralleling models of gravitationally-induced collapse [38].

1.3 The Case for Measurement as a Physical Field

The evidence for measurement as a field phenomenon is both empirical and theoretical. We present seven key lines of evidence:

1.3.1 Weak Measurement and Field Gradients

Weak measurement experiments have reliably shown that the system or particle being measured collapses proportional to the input of measurement-demonstrating that measurement is not a binary force as Copenhagen suggested, but acts on a gradient, one of the base qualifications for a field.

Le's work on 'Phonon-assisted Casimir interactions between piezoelectric materials' demonstrates how Casimir forces can be modulated through material properties, supporting that the propagation of measurement extends beyond the initial point of application [32]. The phonon-assisted interactions show field-like behavior where quantum vacuum fluctuations couple with material excitations. This is further corroborated by Zhang's Magnetic field tuning of the Casimir force" which demonstrates that Casimir forces can be actively modulated by external fields, showing threshold behavior where magnetic field strength creates phase-like transitions [51].

The ability to tune Casimir interactions through external fields provides direct evidence for the field nature of measurement. As Stange et al. note in their comprehensive review of Casimir effect science and technology, these vacuum fluctuation effects are not mere theoretical curiosities but measurable phenomena with technological applications [41].

1.3.2 The Zeno Effect as Field Accumulation

The Zeno Effect provides empirical evidence that measurement operates as a gradient field. With repeated measurement, there is an enforced stable state-experiments confirm that continued measurement exerts enough pressure to maintain stability. This persistence of effects, increasing with measurement field intensity, demonstrates exactly how field interactions accumulate over time.

Recent work on speeding up quantum measurement using space-time trade-offs further supports this view, showing that measurement efficiency can be optimized through field-like manipulation of observation geometry [12].

What's more is that the Zeno effect seemingly is used during negative temperature testing, (the actual negative, where cold flows to hot) as the effects of measurement through laser refraction keep the target from evolving. [50, 11, 31]

1.3.3 Virtual Particles and Measurement Bleed

The existence of virtual particles shows a stepping behavior where intense measurement propagates across nearby systems into lower measurement areas-implying that systems themselves bleed measurement intensity. In the same manner as magnetic fields propagate beyond their initial point of application, the measurement field propagates beyond theirs. This explains why virtual particles appear in close proximity to intense measurement events rather than randomly throughout space.

Jaeger, in his writing on Virtual particles, "Are Virtual Particles Less Real," states that

"Accordingly, any particle not eventually appearing as a quantum of a state of any free field is virtual. However, as noted, for many, if not all, sorts of particle that can appear as a free particle, there are circumstances in which that particle can appear as a virtual particle (i.e., a quantum associated with a distinct, mediating field). Therefore, the distinction is not a fundamental one and any objection on this basis to Position IV fails."[29]

This effectively dismantles the semantic argument that virtual particles are not real. Not only do they possess ontological validity, but their existence mediates interactions across space and time without adhering strictly to classical constraints, as they function as propagators within the quantum field. Within a framework defined by measurement or collapse, such as the one proposed here, virtual particles represent precisely the class of nonlocal, decoherence-spanning entities that only acquire definition when the full observational context has been resolved.

Nakata and Suzuki's work on the 'Non-Hermitian Casimir Effect of Magnons" provides compelling evidence for this measurement bleed phenomenon, showing how magnon interactions create non-Hermitian effects that alter vacuum structure [37]. Similarly, computational modeling of the semi-classical quantum vacuum in 3D by Zhang et al. reveals structured patterns in vacuum fluctuations that align with measurement field predictions [53].

1.3.4 Quantum Entanglement and Non-Local Field Structure

The nature of quantum entanglement demonstrates that fields propagate over distance. Two entangled particles maintain correlations across great distances, suggesting a field-like structure that isn't constrained by spatial limits. This supports the measurement field as a non-local phenomenon with instant propagation characteristics.

Khatiwada and Qian's work on 'Wave-Particle Duality Ellipse and Application in Quantum Imaging with Undetected Photons" demonstrates how quantum correlations can be exploited without direct measurement, supporting the field-like propagation of measurement influence [30].

1.3.5 Phase Transition Behavior

Systems exhibit sharp transitions from quantum to classical behavior when measurement interaction exceeds certain thresholds. This isn't gradual but exhibits the sudden state change characteristic of phase transitions. Enkner's work on 'Tunable vacuum-field control of fractional and integer quantum Hall phases" shows how vacuum fields can control phase transitions between quantum states, directly supporting the phase-like transition of measurement [20].

The observation of negative temperature states in optical lattices offers compelling evidence for phase-like transitions driven by measurement constraints. Braund et al. and Donini et al. independently demonstrated the emergence of quantum phases in frustrated triangular and Kagome lattice configurations at negative absolute temperatures, where the system's phase structure is dictated not solely by energy minimization, but by the geometric tension of measurement itself [11]. [19] These negative bosonic states arise from an enforced measurement geometry that exceeds the definitional capacity of the system, generating "impossible" configuration spaces. This directly correlates negative temperature with measurement field intensity, implying that thermodynamic inversion is not a statistical anomaly, but a structural collapse response under definitional overload.

1.3.6 Temporal Reflection and Time-Domain Collapse

Moussa et al.'s groundbreaking 'Observation of temporal reflection and broadband frequency translation at photonic time interfaces" demonstrates that electromagnetic waves can be reflected in time rather than space [36]. This temporal interface behavior strongly supports the MFT view that time itself emerges from measurement field dynamics, with temporal boundaries acting as collapse surfaces.

1.3.7 Interference Patterns and Classical Emergence

Villas-Boas et al.'s work on 'Bright and Dark States of Light: The Quantum Origin of Classical Interference" reveals how quantum superposition states give rise to classical interference patterns through measurement interaction [45]. This provides a direct bridge between quantum potential and classical observation, supporting the MFT framework where classical states emerge from measurement field interaction.

1.3.8 Dark Matter as Unmeasured Coherent State

In Measurement Field Theory, dark matter represents a state of macroscopic quantum coherence-resistant to measurement. Liang and Caldwell's proposal that 'Cold Dark Matter Based on an Analogy with Superconductivity" aligns deeply with this view, where dark matter is a coherent quantum state like superconductivity. Liang's physics perspective on this work emphasizes how superconductivity inspires new dark matter models [33, 54].

The odd gravitational clustering patterns in dwarf galaxies reflect the non-standard gravitational effects of dark matter as unmeasured potential [52]. Additional evidence comes from reports of 'strange 'sticky' dark matter" that could be lurking in distant galaxies, exhibiting behavior consistent with coherent, measurement-resistant states [21].

The possibility of black holes growing inside stars, as explored by Chakraborty, suggests another manifestation of measurement-resistant states where collapse occurs in isolation from external observation [1].

1.4 Field Genesis and Collapse Dynamics

At the core of MFT lies the imaginary-real dual nature of potential:

$$M(x,t) = A(x) + iB(x,t) \tag{1.2}$$

where A is the observable real projection and B is the imaginary potential reservoir.

Collapse occurs through rotational phase decay:

$$\theta(x,t) = \arctan\left(\frac{B(x,t)}{A(x)}\right) \tag{1.3}$$

The angular phase velocity defines local collapse time:

$$\frac{d\theta}{dt} = -\frac{\alpha A(x)B(x,t)}{A^2(x) + B^2(x,t)} \tag{1.4}$$

with collapse-dependent chronology given by:

$$T(x,t) = \int_0^t \frac{d\theta}{d\tau}\, d\tau \tag{1.5}$$

This framework replaces absolute time with collapse-relative evolution.

1.5 Imaginary Matrices in Three-Dimensional Realspace

The magnitude of the combined field is:

$$|M| = \sqrt{A^2 + B^2}.$$

To build physical intuition in three spatial dimensions, we embed each matrix element $M_{ij} = a_{ij} + ib_{ij}$ into $\mathbb{R}^3$ by:

$$(i, j, a_{ij}) \mapsto \begin{cases} \text{height} = a_{ij}, & \text{(classical elevation)} \\ \text{hue/opacity} \propto |b_{ij}|, & \text{(imaginary intensity)} \\ \text{vector spin angle} \propto \arg(b_{ij}), & \text{(phase rotation)} \end{cases}$$

This visualization directly analogizes quantum-state tomography but with a real-space scaffold.

1.6 Collapse Dynamics: Temporal and Spatial Evolution

Collapse is treated as a temporal decay process governed by:

$$\frac{\partial B}{\partial t} = -\alpha B,$$

leading to the solution:

$$B(\vec{x}, t) = B_0(\vec{x}) e^{-\alpha t}.$$

The resulting dynamics for the field magnitude are:

$$\frac{\partial |M|}{\partial t} = -\alpha \frac{B^2}{\sqrt{A^2 + B^2}} \tag{1.6}$$

$$\nabla |M| = \frac{A \nabla A + B \nabla B}{\sqrt{A^2 + B^2}} \tag{1.7}$$

Theorem 1.6.1 (Collapse Gradient Theorem)**.** *The temporal decay* (1.6) *and spatial tension* (1.7) *completely characterize first-order collapse flow in MFT.*

As $B \to 0$, $\partial_t |M| \to 0$-the field has finished snapping.

Onboarding: The Hessian Hazing Ritual

Okay, problem children, we have a new student.

The good news? You can use Hessians to define their relevance in math. Second derivatives, eigenvalue analysis, critical point classification-delicious tools for the discerning theorist.

The bad news? They're part of a Heaviside function that defines reality. One slip, one sign error, one moment of neglect, and **Bakugo's fingers are gone.**

> *"Reality doesn't have training wheels. It has discontinuities."*

We're not in Calculus I anymore, Toto. We're differentiating piecewise functions that would make Gauss drink. So bring your gradients, bring your grit, and remember:

Symmetry won't save you. This will be further examined through retrocausality in subsequent work, if the universe is still accepting manuscripts by then.

1.7 Collapse Geometry and Quantum Field Coupling

1.7.1 Collapse Curvature Tensor

Collapse curvature is embedded in second spatial derivatives:

$$\Gamma_{ij}(x,t) = \partial_i \partial_j M(x,t) \tag{1.8}$$

This tensor serves as a geometric encoding of collapse stress, deforming the fabric of spacetime and interacting with quantum symmetry fields. Collapse stress gradients can be seen as sources of coherence amplification or decoherence, depending on the alignment of observers and local entropy density.

1.7.2 Quantum Chromodynamics via Collapse Projection

The gluon field strength tensor $G^a_{\mu\nu}$ emerges as a projection of collapse curvature gradients:

$$G^a_{\mu\nu} = f^a_{ij} \Gamma_{ij} \tag{1.9}$$

where f^a_{ij} are projection coefficients mapping collapse tensor components to SU(3) colour space.

Effective QCD Lagrangian:

$$\mathcal{L}^{\text{Collapse}}_{QCD} = -\frac{1}{4} f^a_{ij} f^a_{kl} \Gamma_{ij} \Gamma_{kl} + \bar{\psi}_i (i\gamma^\mu D_\mu - m_i)\psi_i \tag{1.10}$$

This treats gluons as field topology gradients under observer-defined symmetry projection. SU(3) emerges not as a fundamental symmetry but as a preferred projection geometry from measurement collapse structure.

1.7.3 General Relativity via Collapse Ricci Tensor

Define the collapse Ricci tensor:

$$R^{\text{collapse}}_{ij} = \partial_i \partial_j M - \Box M \delta_{ij} \tag{1.11}$$

where $\Box M$ is the d'Alembertian:

$$\Box M = \frac{\partial^2 M}{\partial t^2} - \nabla^2 M \tag{1.12}$$

Einstein tensor emergence:

$$G_{ij} = R^{\text{collapse}}_{ij} - \frac{1}{2} g_{ij} \sum_k R^{\text{collapse}}_{kk} \tag{1.13}$$

This implies spacetime curvature is the macroscopic aggregate of collapse Ricci tensors under coherent observer density. The presence of $\Box M$ links relativistic propagation and collapse evolution.

1.8 Collapse Field Action and Unified Dynamics

To formalize collapse dynamics, we introduce a Lagrangian density:

$$\mathcal{L} = \frac{1}{2}(\partial_\mu M^* \partial^\mu M) - V(|M|) + \mathcal{L}_{\text{obs}} + \mathcal{L}_{\text{collapse}},$$

where the collapse potential drives $B \to 0$:

$$V(|M|) = \frac{1}{2}\alpha^2 B^2.$$

The total action:

$$S[M] = \int \mathcal{L}(M, \partial_\mu M, g_{\mu\nu}, \rho_{\text{obs}})\sqrt{-g}\, d^4x,$$

yields field equations through variational principle:

$$\Box M + \frac{dV}{dM} = \text{Observer Terms.}$$

This leads to our fundamental Collapse Law Alpha (Symbolic Form):

$$\boxed{\mathcal{C} = \Box M + \nabla^2 M + \Theta = 0} \tag{1.14}$$

where:

- $\Box M$: d'Alembertian (temporal-spatial collapse curvature)
- $\nabla^2 M$: Laplacian (spatial definition diffusion)
- Θ: composite observational feedback term-includes imaginary reflux, curvature stress, observational flux, void impedance, resonance harmonics, and annihilation field ratios

This equation serves as the canonical symbolic law for field collapse dynamics, unifying relativistic structure, spatial coherence, and observational influence-our $E = mc^2$.

1.9 Dimensional Consistency of Field Parameters

Parameter	Meaning	Units
D	Collapse diffusivity	L^2/T
α	Temporal decay rate	$1/T$
κ	Observer density coupling	L^3/M
λ	Observer-observer coupling	L^3/M
δ	Relativistic propagation constant	L^2/T^2
μ	Memory integration rate	$1/T$
γ	Memory decay rate	$1/T$
ω_0	Resonance frequency	$1/T$
β	Observer coupling exponent	Dimensionless

1.10 Measurement Entropy and Thermodynamics of Collapse

We define a local entropy density:

$$\mathcal{S}(\mathbf{x}, t) = -\eta B^2(\mathbf{x}, t) \ln \left[\frac{B^2(\mathbf{x}, t)}{B_0^2(\mathbf{x})} \right],$$

paralleling von Neumann entropy but collapsed to a field representation. The total measurement entropy:

$$S(t) = \int \mathcal{S}(\mathbf{x}, t) \, d^3x$$

monotonically decreases, $\dot{S} < 0$, until only classical states remain.

Viewing $B^2(\mathbf{x})$ as an energy landscape, we introduce:

$$Z = \int \exp\left[-\beta B^2(\mathbf{x}) \right] d^3x, \quad \beta = \alpha^{-1}.$$

The normalized field-ensemble probability:

$$P(\mathbf{x}) = \frac{\exp[-\beta B^2(\mathbf{x})]}{Z}$$

reveals measurement as a cooling process: high-B (unresolved) regions are exponentially suppressed.

1.11 Lagrangian Derivation of the Collapse Field Equation

We define the collapse field $M(x^\mu)$ as a scalar field influenced by observer flux, curvature deformation, annihilation gradients, and entropic pressures. The full dynamics can be derived from a Lagrangian density using the Euler-Lagrange formalism.

1.11.1 Lagrangian Density

We begin with a relativistic scalar field Lagrangian of the form:

$$\mathcal{L} = \frac{1}{2} \partial^\mu M \, \partial_\mu M - V(M, \partial M, \rho_{\text{obs}}, M_i, \theta, t) \tag{1.15}$$

where:

$$\partial^\mu M \, \partial_\mu M = \frac{1}{c^2} \left(\frac{\partial M}{\partial t} \right)^2 - |\nabla M|^2$$
$$x^\mu = (ct, x, y, z)$$

The potential V includes all field interactions, feedbacks, and nonlinear collapse mechanics.

1.11.2 Euler-Lagrange Equation

We apply the field-theoretic Euler-Lagrange equation:

$$\frac{\partial \mathcal{L}}{\partial M} - \partial_\mu \left(\frac{\partial \mathcal{L}}{\partial(\partial_\mu M)} \right) = 0 \tag{1.16}$$

Substituting the Lagrangian, we obtain:

$$\Box M + \frac{\partial V}{\partial M} = 0 \tag{1.17}$$

where the d'Alembertian is:

$$\Box M = \frac{1}{c^2}\frac{\partial^2 M}{\partial t^2} - \nabla^2 M \tag{1.18}$$

1.11.3 Collapse Potential

The collapse potential V incorporates observer flux, field memory, curvature deformation, imaginary feedback, annihilation damping, and resonance coupling:

$$\begin{aligned} V(M) = &+ \frac{1}{2}\lambda M^2 \quad \text{(collapse sink)} \\ &- \kappa \frac{\rho_{\text{obs}}(x,t)}{r^2} M \quad \text{(observer injection)} \\ &- \frac{1}{2} H(t) M^2 \quad \text{(entropy inflation)} \\ &+ \xi M \cdot \nabla^2 \rho_{\text{obs}}(x,t) \quad \text{(void damping)} \\ &+ \zeta_{\text{ann}} \left| \nabla \left(\frac{\rho_{\text{matter}} - \rho_{\text{antimatter}}}{\rho_{\text{total}} + \epsilon} \right) \right|^2 \quad \text{(annihilation sink)} \\ &- \chi \cdot \log \left[\cosh \left(\frac{M_i}{M_r + \epsilon} \right) \right] \quad \text{(soft reflux)} \\ &+ \frac{\sigma}{2} \sum_{i,j} (\Gamma_{ij})^2 \quad \text{(collapse tensor stress)} \\ &- \nu \cdot \cos(2\omega_0 t - 2\theta(x,t)) M \quad \text{(resonance modulation)} \\ &+ \zeta \cdot \eta(x,t) M \quad \text{(noise injection)} \\ &+ \frac{\mu}{2} \left(\int_{t_0}^{t} M(\tau) e^{-\gamma(t-\tau)} d\tau \right)^2 \quad \text{(memory kernel)} \end{aligned} \tag{1.19}$$

1.11.4 Derived Collapse Field Equation

Inserting the potential into the Euler-Lagrange formalism yields:

$$\boxed{\Box M + \lambda M - \kappa \frac{\rho_{\text{obs}}}{r^2} - H(t) M + \xi \nabla^2 \rho_{\text{obs}} - \nu \cos(2\omega_0 t - 2\theta) + \cdots = 0} \tag{1.20}$$

where the omitted terms arise from derivatives of the remaining nonlinear potential components.

1.11.5 Collapse Tensor Definition

The collapse curvature tensor Γ_{ij} is defined as:

$$\Gamma_{ij} = \frac{\partial^2 M}{\partial x_i \partial x_j} - \frac{1}{3}\delta_{ij}\nabla^2 M \tag{1.21}$$

and contributes through its Frobenius norm:

$$\sum_{i,j}(\Gamma_{ij})^2 = ||\Gamma||^2 \tag{1.22}$$

1.11.6 Collapse Law Alpha (Unified Field Form)

The full evolution is governed by:

$$\boxed{\mathcal{C} = \Box M + \nabla^2 M - \lambda M + \frac{\rho_{\text{obs}}}{r^2} + \Phi_{\text{imag}} + \Sigma_{\text{curv}} + \Psi_{\text{void}} + \Omega_{\text{res}} = 0} \tag{1.23}$$

1.12 Hamiltonian Formalism of Collapse Field Dynamics

To enable phase-space simulations, canonical quantization, and derivation of conserved quantities, we now express the collapse field theory in Hamiltonian form.

1.12.1 Canonical Momentum

Starting with the Lagrangian density:

$$\mathcal{L} = \frac{1}{2c^2}\left(\frac{\partial M}{\partial t}\right)^2 - \frac{1}{2}|\nabla M|^2 - V(M, \nabla M, \rho_{\text{obs}}, M_i, t) \tag{1.24}$$

we define the canonical conjugate momentum:

$$\pi(x,t) = \frac{\partial \mathcal{L}}{\partial(\partial_t M)} = \frac{1}{c^2}\frac{\partial M}{\partial t} \tag{1.25}$$

1.12.2 Hamiltonian Density

The Hamiltonian density is constructed via Legendre transform:

$$\mathcal{H} = \pi\frac{\partial M}{\partial t} - \mathcal{L} \tag{1.26}$$

Substituting $\partial_t M = c^2\pi$, we obtain:

$$\mathcal{H}(M, \pi, x, t) = \frac{1}{2}c^2\pi^2 + \frac{1}{2}|\nabla M|^2 + V(M, \nabla M, \rho_{\text{obs}}, M_i, t) \tag{1.27}$$

1.12.3 Canonical Equations of Motion

The first-order Hamiltonian equations governing collapse dynamics are:

$$\frac{\partial M}{\partial t} = \frac{\delta \mathcal{H}}{\delta \pi} = c^2\pi \tag{1.28}$$

$$\frac{\partial \pi}{\partial t} = -\frac{\delta \mathcal{H}}{\delta M} = -\left(-\nabla^2 M + \frac{\partial V}{\partial M}\right) \tag{1.29}$$

Combining yields the second-order collapse evolution equation:

$$\frac{\partial^2 M}{\partial t^2} = c^2\left(\nabla^2 M - \frac{\partial V}{\partial M}\right) \tag{1.30}$$

1.12.4 Collapse Hamiltonian Structure

The total Hamiltonian encodes the energy content of the collapse field, including:

- Kinetic collapse energy $\frac{1}{2}c^2\pi^2$
- Spatial definitional diffusion $\frac{1}{2}|\nabla M|^2$
- Nonlinear collapse dynamics $V(M, \nabla M, ...)$

This form enables:

- Phase-space simulation of collapse dynamics
- Canonical quantization using Poisson brackets or path integrals
- Derivation of the energy-momentum tensor via Noether's theorem

1.12.5 Optional: Energy-Momentum Tensor

The stress-energy tensor for the collapse field may be derived as:

$$T^{\mu\nu} = \frac{\partial \mathcal{L}}{\partial(\partial_\mu M)}\partial^\nu M - \eta^{\mu\nu}\mathcal{L} \tag{1.31}$$

This yields conserved energy and momentum fluxes across spacetime domains under Lorentz symmetry.

1.13 Technological Applications and Experimental Directions

The practical implications of MFT extend beyond theoretical physics. Lander Gower et al.'s work on molecular beam epitaxy growth characteristics for terahertz quantum cascade lasers demonstrates how measurement field control at the quantum level can optimize device performance [31]. Similarly, Yang et al.'s development of thermoelectric porous laser-induced graphene-based strain-temperature decoupling shows how measurement field principles can enable self-powered sensing [50].

1.14 From Quantum to Classical: The Collapse Mechanism

The transition from quantum superposition to classical reality occurs through measurement-induced collapse. This is not merely a theoretical abstraction-it is the only empirically observed mechanism for wavefunction collapse. As established in our field equations, collapse propagates as a physically real phenomenon exhibiting:

- Gradient behavior (weak measurement signatures)
- Field-like propagation (Casimir forces and virtual particle interactions)
- Threshold transitions (observable phase-change events)
- Non-local entanglement correlations
- Entropy reduction under over-defined conditions
- Temporal boundary effects (e.g., quantum time-reflection phenomena)
- Practical manifestation in quantum devices (e.g., cascade lasers, sensors)

1.15 Implications and Testable Predictions

The Measurement Field framework yields several empirically testable predictions:

1. Casimir cavity configurations should reveal detectable variations in measurement field intensity.
2. Phase transitions in quantum systems should correspond to critical measurement thresholds.
3. Dark matter regions will exhibit suppressed or delayed collapse field signatures.
4. Virtual particle density should follow observable measurement field gradients.

5. Decoherence rates will vary with local measurement field saturation.
6. Temporal interfaces (e.g., time crystals or reflection events) will demonstrate collapse field echo or rebound.
7. Quantum device efficiency will peak at specific collapse geometry configurations.

1.16 Currently Untestable but Expected Predictions

We further propose several predictions currently beyond technological validation, but logically extrapolated from the collapse field framework:

1. Black hole cores are dominated by extreme measurement field intensity, with potential flowing outward via recursive over-definition.
2. Dark matter exists in large-scale quantum superposition, especially in low-definition galactic fringe regions.
3. Dark flows consist of superheavy elements traveling along low-potential collapse corridors-particularly near voids or galactic boundaries.
4. Neutron stars are the fossilized remnants of black holes, bled into stable high-density collapse equilibrium by potential exhaustion.
5. Virtual particles are undetectable in low-measurement regions but become increasingly defined under intense observational pressure-up to and including negative temperature inversion states.
6. The fundamental interaction hierarchy follows collapse definition thresholds: gravity emerges first under low-definition, followed by the strong and weak nuclear forces, and finally electromagnetism in high-definition zones.
7. The speed of definition (collapse propagation) must exceed the speed of light-potentially by a factor of at least two-to explain the observed coherence of quantum states across spacetime.

1.17 Conclusion

Through the synthesis of imaginary matrices, Euler's identity as collapse operator, and empirical evidence for measurement as a field, we have constructed a unified framework for understanding quantum-classical transitions. Measurement is not an abstract postulate but a physical field with definable characteristics, propagation laws, and thermodynamic properties.

Measurement Field Theory fundamentally redefines the architecture of reality by treating time as a potential dimension, not a linear progression. Using Euler's identity as the backbone of dimensional transition, MFT collapses infinite regress and divergence into a recursive, finite structure. This not only disarms the infinities that force renormalization in conventional theories, but provides a physically and philosophically coherent framework in which each observable state contains, contextualizes, and completes the history of all lower dimensions. In this paradigm, infinities are not problems to be solved, but symptoms of an outdated conceptual model.

The Collapse Law Alpha- $\mathcal{C} = \Box M + \nabla^2 M + \Theta = 0$ -captures in symbolic form what decades of quantum mechanics have struggled to articulate: that reality emerges from the interplay of spacetime curvature, spatial coherence, and observational influence. This is not just a theory of measurement; it is a theory of how the universe continuously creates itself through the act of definition.

As measurement forces the imaginary to become real, potential to become actual, and uncertainty to become structure, we see that collapse is not a mystery to be solved but a fundamental process to be understood. It is the engine of reality itself.

1.18 The Lilith Simulation: Tensor Morphogenesis and Fractal Collapse Shells

To support the proposed field-theoretic framework of measurement collapse, we implemented a numerical model named **Lilith 1.0**, a GPU-accelerated simulation of imaginary-real tensor fields. This system evolves observer-driven measurement collapse across hierarchical shell layers in 3D realspace.

1.18.1 Simulation Framework

The simulation evolves a scalar field $M(x, y, z, t)$ using the following second-order update equation:

$$M_{t+1} = 2M_t - M_{t-1} + \Delta t^2 \left(c^2 D \nabla^2 M - \lambda M + \kappa \rho_{\text{obs}} \right), \tag{1.32}$$

where ρ_{obs} is the observer density field, D is diffusivity, λ is decay rate, and κ is observer coupling. Observer agents traverse the field, replicating in coherent regions and modifying ρ_{obs} dynamically. Imaginary components M_i are used to bias drift, producing agent trajectories influenced by collapse potential.

1.18.2 Observer Drift Algorithm

Observer positions are updated using gradient-following motion:

$$\vec{v} = (1 - \gamma)\nabla M + \gamma \nabla M_i, \tag{1.33}$$

with additional cohesion and mobility decay applied. Replication occurs when $|M - M_{\text{prev}}| < \epsilon$, and agents avoid shell boundaries via potential feedback.

1.18.3 Full Code Listing

`Lilith1.0.py` is included below for full reproducibility. Please note that some functions are disabled pending futher testing.

```
import os
os.environ["CUPY_NVCC_GENERATE_CODE"] = "--std=c++17"
import tkinter as tk
from tkinter import ttk, filedialog, messagebox
import threading
import queue
import time
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.figure import Figure
import json
from datetime import datetime
import random
import warnings
warnings.filterwarnings('ignore')  # Suppress matplotlib warnings

import cupy as cp
import healpy as hp
from scipy.signal import correlate
from scipy.special import rel_entr
from cupyx.scipy.ndimage import convolve

from healpy.sphtfunc import map2alm, alm2cl
# Import the actual simulation modules
try:
    CUPY_AVAILABLE = True
    print("CuPy and HEALPix successfully imported")
except ImportError as e:
    print(f"Warning: CuPy/HEALPix not available, using NumPy fallback: {e}")
    import numpy as cp
    CUPY_AVAILABLE = False

    # Mock HEALPix functions for fallback
    class MockHealPy:
        @staticmethod
        def nside2npix(nside):
            return 12 * nside * nside
        @staticmethod
        def ang2pix(nside, theta, phi):
            return np.zeros(len(theta), dtype=int)
        @staticmethod
        def read_map(filename, **kwargs):
            return np.random.random(12 * 256 * 256)
        @staticmethod
        def ud_grade(map_in, nside_out):
            return np.random.random(12 * nside_out * nside_out)
        @staticmethod
        def anafast(map_in, **kwargs):
            return np.random.random(500)

    hp = MockHealPy()

```

```
    def map2alm(map_in, **kwargs):
        return np.random.random(500) + 1j * np.random.random(500)

    def alm2cl(alm):
        return np.random.random(len(alm))

    try:
        from scipy.special import rel_entr
    except ImportError:
        def rel_entr(p, q):
            return p * np.log(p / q)

class LilithSimulation:
    """Core simulation class integrating Lilith 1.0 with real-time analysis"""
    def make_gravity_kernel(self):
        kernel = cp.zeros((3, 3, 3), dtype=cp.float32)
        center = cp.array([1, 1, 1])

        for x in range(3):
            for y in range(3):
                for z in range(3):
                    pos = cp.array([x, y, z])
                    dist = cp.linalg.norm(pos - center)
                    if dist > 0:
                        kernel[x, y, z] = 1.0 / (dist**2)
        kernel /= cp.sum(kernel)  # Normalize to prevent runaway force
        return kernel

    def __init__(self, params, output_queue, custom_output_dir=None):
        self.params = params
        self.output_queue = output_queue
        self.running = False
        self.step = 0
        self.custom_output_dir = custom_output_dir
        self.files_saved_count = 0
        self.gravity_kernel = self.make_gravity_kernel()

        # Create output directory
        self.setup_output_directory()

        # Initialize simulation state
        self.initialize_simulation()

        # Load Planck data for comparison
        self.load_planck_data()
    import cupy as cp


    def setup_output_directory(self):
        """Create output directory for saving results"""
        if self.custom_output_dir:
            timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
            self.output_dir = os.path.join(self.custom_output_dir,
    f"lilith_run_{timestamp}")
```

```python
        else:
            timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
            self.output_dir = f"lilith_gui_output_{timestamp}"

        os.makedirs(self.output_dir, exist_ok=True)
        print(f"Output directory created: {self.output_dir}")

    def initialize_simulation(self):
        """Initialize the simulation with current parameters"""
        size = self.params['size']
        max_layers = self.params['max_layers']
        n_obs = self.params['n_obs']

        # Initialize layers
        self.M_layers = []
        self.M_prev_layers = []
        self.M_i_layers = []
        self.rho_obs_layers = []
        self.shell_masks = []
        self.shell_surfaces = []
        self.radius_shells = []
        self.observer_states = []
        self.nucleation_fields = []
        self.memory_fields = []

        # Generate fractal layers
        for i in range(max_layers):
            scale = self.params['shell_scale_factor'] ** i
            center = size // 2
            xg, yg, zg = cp.meshgrid(cp.arange(size), cp.arange(size), cp.arange(size), indexing='ij')
            dx, dy, dz = xg - center, yg - center, zg - center
            radius_grid = cp.sqrt(dx**2 + dy**2 + dz**2)
            radius_shell = radius_grid.astype(cp.int32)
            shell_max = int(radius_grid.max() * scale)
            mask = (radius_grid <= shell_max).astype(cp.float32)
            surface = ((radius_grid >= shell_max - 1.5) & (radius_grid <= shell_max)).astype(cp.float32)

            M = self.white_noise_field((size, size, size)) * 0.1 * (1.0 / (1 + i))
            M_prev = M.copy()
            M_i = self.white_noise_field((size, size, size), scale=0.001)
            rho_obs = cp.zeros_like(M)

            # Initialize observers
            ob_x = cp.random.randint(0, size, n_obs)
            ob_y = cp.random.randint(0, size, n_obs)
            ob_z = cp.random.randint(0, size, n_obs)
            ob_age = cp.zeros(n_obs, dtype=cp.int32)
            ob_fn = cp.zeros(n_obs, dtype=cp.int32)
            ob_alive = cp.ones(n_obs, dtype=cp.bool_)
            ob_mob = cp.ones(n_obs, dtype=cp.float32)

            self.M_layers.append(M * mask)
```

```
            self.M_prev_layers.append(M_prev * mask)
            self.M_i_layers.append(M_i * mask)
            self.rho_obs_layers.append(rho_obs)
            self.radius_shells.append(radius_shell)
            self.shell_masks.append(mask)
            self.shell_surfaces.append(surface)
            self.observer_states.append({
                "x": ob_x, "y": ob_y, "z": ob_z, "age": ob_age,
                "fn": ob_fn, "alive": ob_alive, "mobility": ob_mob
            })
            self.nucleation_fields.append(cp.zeros_like(M))
            self.memory_fields.append(cp.zeros_like(M))

        # Store grid coordinates for projections
        self.dx, self.dy, self.dz = dx, dy, dz

    def white_noise_field(self, shape, scale=0.1):
        """Generate white noise field"""
        noise = cp.random.normal(loc=0.0, scale=scale, size=shape)
        freq_noise = cp.fft.fftn(noise)
        random_phase = cp.exp(2j * cp.pi * cp.random.rand(*shape))
        filtered = cp.real(cp.fft.ifftn(freq_noise * random_phase))
        return filtered

    def laplacian_3d(self, F):
        """3D Laplacian operator"""
        return (
            cp.roll(F, 1, axis=0) + cp.roll(F, -1, axis=0) +
            cp.roll(F, 1, axis=1) + cp.roll(F, -1, axis=1) +
            cp.roll(F, 1, axis=2) + cp.roll(F, -1, axis=2) -
            6 * F
        )

    def observer_drift(self, M, ob, radius_shell, shell_max):
        """Observer movement and dynamics"""
        pot = M + 0.5 * self.laplacian_3d(M)
        grad_x, grad_y, grad_z = cp.gradient(pot)
        gx = grad_x[ob["x"], ob["y"], ob["z"]]
        gy = grad_y[ob["x"], ob["y"], ob["z"]]
        gz = grad_z[ob["x"], ob["y"], ob["z"]]
        norm = cp.sqrt(gx**2 + gy**2 + gz**2) + 1e-6

        ob["mobility"] *= self.params['observer_mobility_decay']

        # Cohesion behavior
        x_c, y_c, z_c = ob["x"], ob["y"], ob["z"]
        x_mean, y_mean, z_mean = cp.mean(x_c), cp.mean(y_c), cp.mean(z_c)
        cx = x_mean - x_c
        cy = y_mean - y_c
        cz = z_mean - z_c
        c_norm = cp.sqrt(cx**2 + cy**2 + cz**2) + 1e-6
        cohesion_weight = 0.9
        gx = (1 - cohesion_weight) * gx + cohesion_weight * (cx / c_norm)
        gy = (1 - cohesion_weight) * gy + cohesion_weight * (cy / c_norm)
```

```
        gz = (1 - cohesion_weight) * gz + cohesion_weight * (cz / c_norm)

        norm = cp.sqrt(gx**2 + gy**2 + gz**2) + 1e-6
        step_size = self.params.get('step_size', 0.5)
        size = self.params['size']

        x_new = cp.clip(ob["x"] + ob["mobility"] * step_size * (gx / norm), 0, size - 1).astype(cp.int32)
        y_new = cp.clip(ob["y"] + ob["mobility"] * step_size * (gy / norm), 0, size - 1).astype(cp.int32)
        z_new = cp.clip(ob["z"] + ob["mobility"] * step_size * (gz / norm), 0, size - 1).astype(cp.int32)

        # Handle shell boundaries
        r_obs = radius_shell[x_new, y_new, z_new]
        shell_hit = (r_obs >= shell_max)
        x_new[shell_hit] = size // 2
        y_new[shell_hit] = size // 2
        z_new[shell_hit] = size // 2

        return x_new, y_new, z_new

    def load_planck_data(self):
        """Load Planck CMB data for comparison"""
        try:
            # Try to load local Planck data - check multiple possible filenames
            planck_fits_files = ["SMICA_CMB.FITS", "smica_cmb.fits", "COM_CMB_IQU-smica_1024_R2.02_full.fits"]
            planck_cl_files = ["COM_PowerSpect_CMB-TT-full_R3.01.txt", "planck_2018_cls.txt"]

            self.planck_map = None
            self.planck_cl = None

            # Try to load FITS file
            for fname in planck_fits_files:
                if os.path.exists(fname):
                    try:
                        print(f"Loading Planck map from {fname}")
                        self.planck_map = hp.read_map(fname, field=0, verbose=False)
                        self.planck_map = hp.ud_grade(self.planck_map, nside_out=self.params['nside'])
                        print(f"Successfully loaded Planck map with nside={self.params['nside']}")
                        break
                    except Exception as e:
                        print(f"Failed to load {fname}: {e}")
                        continue

            # Try to load power spectrum file
            for fname in planck_cl_files:
                if os.path.exists(fname):
                    try:
                        print(f"Loading Planck power spectrum from {fname}")
```

```
                        data = np.loadtxt(fname)
                        self.planck_cl = data[:, 1] if data.shape[1] > 1 else data
                        print(f"Successfully loaded Planck Cl with {len(self.planck_cl)} multipoles")
                        break
                    except Exception as e:
                        print(f"Failed to load {fname}: {e}")
                        continue

            # Generate Planck Cl from map if we have map but no Cl file
            if self.planck_map is not None and self.planck_cl is None:
                try:
                    print("Generating power spectrum from Planck map...")
                    self.planck_cl = hp.anafast(self.planck_map, lmax=min(512, 3*self.params['nside']-1))
                    print(f"Generated Planck Cl with {len(self.planck_cl)} multipoles")
                except Exception as e:
                    print(f"Failed to generate Cl from map: {e}")

        except Exception as e:
            print(f"Warning: Could not load Planck data: {e}")
            self.planck_map = None
            self.planck_cl = None

    def compute_metrics(self):
        """Compute real-time analysis metrics"""
        metrics = {}

        # Combine shell data for projection
        size = self.params['size']
        combined_shell = cp.zeros((size, size, size))
        for i in range(len(self.M_layers)):
            combined_shell += self.M_layers[i] * self.shell_surfaces[i]

        # Convert to HEALPix projection
        shell_energy = float(cp.sum(combined_shell))
        metrics['shell_energy'] = shell_energy

        if shell_energy > 1e-6:
            # Create HEALPix projection
            r_grid = cp.sqrt(self.dx**2 + self.dy**2 + self.dz**2) + 1e-6
            valid_mask = combined_shell > 0

            if cp.sum(valid_mask) > 0:
                dz_valid = self.dz[valid_mask]
                dy_valid = self.dy[valid_mask]
                dx_valid = self.dx[valid_mask]
                r_valid = r_grid[valid_mask]
                theta = cp.arccos(dz_valid / r_valid)
                phi = cp.arctan2(dy_valid, dx_valid) % (2 * cp.pi)
                weights = combined_shell[valid_mask]

                # Convert to numpy for HEALPix
                theta_np = cp.asnumpy(theta)
```

```
            phi_np = cp.asnumpy(phi)
            weights_np = cp.asnumpy(weights)

            npix = hp.nside2npix(self.params['nside'])
            pix = hp.ang2pix(self.params['nside'], theta_np, phi_np)
            proj = np.bincount(pix, weights=weights_np, minlength=npix)

            # Compute power spectrum
            if np.std(proj) > 1e-6:
                try:
                    alm = map2alm(proj, lmax=min(256, self.params['nside']))
                    cl = alm2cl(alm)

                    # Normalize for entropy calculation
                    # Normalize for entropy calculation
                    cl_norm = cl / (np.sum(cl) + 1e-12)

                    # Compare with Planck if available
                    if self.planck_cl is not None and len(cl) > 10:
                        planck_truncated = self.planck_cl[:len(cl)] / 1e3
                        planck_norm = planck_truncated / (np.sum(planck_truncated) + 1e-12)

                        # Clip both distributions to avoid log(0) fuckery
                        eps = 1e-12
                        cl_norm = np.clip(cl_norm, eps, 1.0)
                        planck_norm = np.clip(planck_norm, eps, 1.0)

                        # KL divergence
                        kl_div = np.sum(rel_entr(cl_norm, planck_norm))
                        metrics['kl_divergence'] = float(kl_div) if not np.isnan(kl_div) else 0.0

                        # Correlation
                        corr = np.corrcoef(cl, planck_truncated)[0, 1]
                        metrics['correlation'] = float(corr) if not np.isnan(corr) else 0.0
                    else:
                        metrics['kl_divergence'] = 0.0
                        metrics['correlation'] = 0.0


                    # Entropy (now cl_norm is always defined)
                    entropy = -np.sum(cl_norm * np.log(cl_norm + 1e-12))
                    metrics['entropy'] = float(entropy)

                    # Store projection for visualization
                    self.current_projection = proj
                    self.current_cl = cl

                    # Save data every 10 steps
                    if self.step % 10 == 0:
                        self.save_projection_data(proj)

```

```
                        # Save power spectrum comparison every 50 steps
                        if self.step % 50 == 0:
                            self.save_power_spectrum_comparison(cl)

                    except Exception as e:
                        print(f"Error computing power spectrum at step {self.step}: {e}")
                        metrics['kl_divergence'] = 0.0
                        metrics['correlation'] = 0.0
                        metrics['entropy'] = 0.0
                else:
                    metrics['kl_divergence'] = 0.0
                    metrics['correlation'] = 0.0
                    metrics['entropy'] = 0.0
            else:
                metrics['kl_divergence'] = 0.0
                metrics['correlation'] = 0.0
                metrics['entropy'] = 0.0
        else:
            metrics['kl_divergence'] = 0.0
            metrics['correlation'] = 0.0
            metrics['entropy'] = 0.0

        # Observer metrics
        total_observers = sum(len(obs["x"]) for obs in self.observer_states)
        metrics['observer_count'] = total_observers

        # Field metrics
        field_variance = float(cp.var(self.M_layers[0]))
        metrics['field_variance'] = field_variance

        # Coherence index
        if len(self.M_layers) > 0:
            coherence = float(cp.mean(cp.abs(self.M_layers[0] - self.M_prev_layers[0])))
            metrics['coherence_index'] = coherence
        else:
            metrics['coherence_index'] = 0.0

        return metrics

    def save_projection_data(self, projection):
        """Save projection data and create Mollweide plot"""
        try:
            # Save NPY file
            npy_filename = os.path.join(self.output_dir,
    f"projection_{self.step:06d}.npy")
            np.save(npy_filename, projection)
            self.files_saved_count += 1

            # Create and save Mollweide plot if HEALPix is available
            if CUPY_AVAILABLE:
                plt.figure(figsize=(12, 6))
                try:
                    hp.mollview(np.log1p(np.abs(projection)),
                               title=f"Lilith Field Collapse - Step {self.step}",
```

```
                               cmap="inferno", cbar=True, hold=True)
                    plot_filename = os.path.join(self.output_dir, f"mollweide_{self.step:06d}.png")
                    plt.savefig(plot_filename, dpi=150, bbox_inches='tight')
                    plt.close()
                    self.files_saved_count += 1
                except Exception as e:
                    print(f"HEALPix mollview error at step {self.step}: {e}")
                    # Fallback to simple plot
                    self.save_simple_projection_plot(projection)
            else:
                self.save_simple_projection_plot(projection)

            # Send file count update to GUI
            self.output_queue.put(('files_saved', {'count': self.files_saved_count}))

        except Exception as e:
            print(f"Error saving projection data at step {self.step}: {e}")

    def save_simple_projection_plot(self, projection):
        """Save a simple 2D projection plot as fallback"""
        try:
            plt.figure(figsize=(10, 8))

            # Create 2D representation
            side_len = int(np.sqrt(len(projection) / 12))
            if side_len < 32:
                side_len = 64

            grid_size = min(128, side_len)
            if len(projection) >= grid_size * grid_size:
                data_2d = projection[:grid_size*grid_size].reshape((grid_size, grid_size))
            else:
                padded_data = np.zeros(grid_size * grid_size)
                padded_data[:len(projection)] = projection
                data_2d = padded_data.reshape((grid_size, grid_size))

            plt.imshow(np.log1p(np.abs(data_2d)), cmap='inferno', aspect='auto')
            plt.colorbar(label='Log(1 + Field Strength)')
            plt.title(f"Lilith Field Projection - Step {self.step}")
            plt.xlabel('Longitude (projected)')
            plt.ylabel('Latitude (projected)')

            plot_filename = os.path.join(self.output_dir, f"projection_2d_{self.step:06d}.png")
            plt.savefig(plot_filename, dpi=150, bbox_inches='tight')
            plt.close()
            self.files_saved_count += 1

        except Exception as e:
            print(f"Error saving simple projection plot at step {self.step}: {e}")

    def save_power_spectrum_comparison(self, cl_data):
```

```
        """Save power spectrum comparison with Planck"""
        try:
            plt.figure(figsize=(12, 8))

            ell = np.arange(len(cl_data))
            plt.loglog(ell[1:], cl_data[1:], label='Lilith Simulation', color='red', linewidth=2)

            # Add Planck comparison if available
            if self.planck_cl is not None:
                planck_truncated = self.planck_cl[:len(cl_data)]
                plt.loglog(ell[1:len(planck_truncated)], planck_truncated[1:],
                          label='Planck 2018 CMB', linestyle='--', color='blue', linewidth=2)

                # Calculate correlation for the plot
                if len(cl_data) > 10:
                    corr = np.corrcoef(cl_data, planck_truncated)[0, 1]
                    plt.text(0.05, 0.95, f'Correlation: {corr:.4f}',
                            transform=plt.gca().transAxes, fontsize=12,
                            bbox=dict(boxstyle="round,pad=0.3", facecolor="white", alpha=0.8))

            plt.xlabel('Multipole moment ', fontsize=14)
            plt.ylabel('C [Kš]', fontsize=14)
            plt.title(f'Angular Power Spectrum Comparison - Step {self.step}', fontsize=16)
            plt.grid(True, alpha=0.3)
            plt.legend(fontsize=12)
            plt.tight_layout()

            plot_filename = os.path.join(self.output_dir, f"power_spectrum_{self.step:06d}.png")
            plt.savefig(plot_filename, dpi=150, bbox_inches='tight')
            plt.close()
            self.files_saved_count += 1

            # Also save the Cl data as NPY
            cl_filename = os.path.join(self.output_dir, f"power_spectrum_{self.step:06d}.npy")
            np.save(cl_filename, cl_data)
            self.files_saved_count += 1

            # Send file count update to GUI
            self.output_queue.put(('files_saved', {'count': self.files_saved_count}))

        except Exception as e:
            print(f"Error saving power spectrum comparison at step {self.step}: {e}")

    def save_final_state(self):
        """Save final simulation state"""
        try:
            # Save final field data
            for i, M_layer in enumerate(self.M_layers):
```

```python
                field_filename = os.path.join(self.output_dir, f"final_field_layer_{i}.npy")
                if hasattr(M_layer, 'get'):  # CuPy array
                    np.save(field_filename, M_layer.get())
                else:  # NumPy array
                    np.save(field_filename, M_layer)

            # Save observer states
            for i, observer_state in enumerate(self.observer_states):
                obs_filename = os.path.join(self.output_dir, f"final_observers_layer_{i}.npy")
                obs_data = {}
                for key, value in observer_state.items():
                    if hasattr(value, 'get'):  # CuPy array
                        obs_data[key] = value.get()
                    else:  # NumPy array
                        obs_data[key] = value
                np.save(obs_filename, obs_data)

            # Save simulation parameters
            params_filename = os.path.join(self.output_dir, "simulation_parameters.json")
            with open(params_filename, 'w') as f:
                json.dump(self.params, f, indent=2)

            print(f"Final simulation state saved to {self.output_dir}")

        except Exception as e:
            print(f"Error saving final state: {e}")

    def simulation_step(self):
        """Execute one simulation step"""
        try:
            size = self.params['size']
            delta_t = self.params['delta_t']
            c = self.params['c']
            D = self.params['D']
            lam = self.params['lam']
            kappa = self.params['kappa']

            for i in range(len(self.M_layers)):
                M, M_prev, M_i, rho_obs = (self.M_layers[i], self.M_prev_layers[i],
                                           self.M_i_layers[i], self.rho_obs_layers[i])
                ob = self.observer_states[i]
                radius_shell = self.radius_shells[i]
                shell_max = int(radius_shell.max())

                # Observer dynamics
                try:
                    ob_x, ob_y, ob_z = self.observer_drift(M, ob, radius_shell, shell_max)
                    ob["x"], ob["y"], ob["z"] = ob_x, ob_y, ob_z
                except Exception as e:
                    print(f"Observer drift error at step {self.step}: {e}")
                    # Skip observer updates but continue with field evolution
```

```

            # Observer replication in coherent zones
            try:
                coherence_zone = cp.abs(M - M_prev) < 0.01
                coherent_indices = cp.where(coherence_zone)
                if len(coherent_indices[0]) > 10:
                    n_new = min(5, len(coherent_indices[0]))
                    if n_new > 0:
                        sampled = cp.random.choice(len(coherent_indices[0]), size=n_new, replace=False)
                        new_x = coherent_indices[0][sampled]
                        new_y = coherent_indices[1][sampled]
                        new_z = coherent_indices[2][sampled]
                        ob["x"] = cp.concatenate((ob["x"], new_x))
                        ob["y"] = cp.concatenate((ob["y"], new_y))
                        ob["z"] = cp.concatenate((ob["z"], new_z))
                        ob["age"] = cp.concatenate((ob["age"], cp.zeros(len(new_x), dtype=cp.int32)))
                        ob["fn"] = cp.concatenate((ob["fn"], cp.zeros(len(new_x), dtype=cp.int32)))
                        ob["alive"] = cp.concatenate((ob["alive"], cp.ones(len(new_x), dtype=cp.bool_)))
                        ob["mobility"] = cp.concatenate((ob["mobility"], cp.ones(len(new_x), dtype=cp.float32)))
            except Exception as e:
                print(f"Observer replication error at step {self.step}: {e}")

            # Update observer density
            try:
                rho_obs *= 0.1
                if len(ob["x"]) > 0:
                    # Ensure indices are valid
                    valid_x = cp.clip(ob["x"], 0, size - 1)
                    valid_y = cp.clip(ob["y"], 0, size - 1)
                    valid_z = cp.clip(ob["z"], 0, size - 1)
                    rho_obs[valid_x, valid_y, valid_z] += 5 * cp.exp(-0.05 * self.step)
            except Exception as e:
                print(f"Observer density update error at step {self.step}: {e}")

            # Field evolution
            try:
                lap = self.laplacian_3d(M)
                decay = -lam * M * float(min(self.step / 5.0, 1.0))
                source = kappa * rho_obs
                accel = c**2 * D * lap + decay + source

                M_next = 2 * M - M_prev + delta_t**2 * accel
                # === GRAVITY CLUMPING ===
                try:
                    gravity_force = convolve(M_next, self.gravity_kernel, mode='reflect')
                    G_strength = 0.005  # you can tweak this like a freak
                    M_next += G_strength * gravity_force
```

```
                    except Exception as e:
                        print(f"Gravity clumping error at step {self.step}: {e}")

                    # Update nucleation fields
                    coherence = cp.abs(M - M_prev)
                    self.nucleation_fields[i] = cp.where((M > 0.05) & (coherence < 0.01), M, 0)
                    self.M_layers[i] = cp.clip(self.M_layers[i], 0.0, 1e3)

                    # Update layers
                    self.M_prev_layers[i] = M
                    self.M_layers[i] = M_next
                    self.M_i_layers[i] = M_i + 0.1 * self.laplacian_3d(M_i) - 0.01 * M_i
                except Exception as e:
                    print(f"Field evolution error at step {self.step}: {e}")
                    # If field evolution fails, try to continue with next layer

            self.step += 1

        except Exception as e:
            print(f"Critical simulation error at step {self.step}: {e}")
            raise  # Re-raise to stop simulation

    def run_simulation(self):
        """Main simulation loop"""
        self.running = True
        start_time = time.time()

        while self.running and self.step < self.params['steps']:
            try:
                self.simulation_step()

                # Compute metrics every 10 steps
                if self.step % 10 == 0:
                    metrics = self.compute_metrics()
                    metrics['step'] = self.step
                    metrics['elapsed_time'] = time.time() - start_time
                    self.output_queue.put(('metrics', metrics))

                # Send visualization data every 50 steps
                if self.step % 50 == 0 and hasattr(self, 'current_projection'):
                    vis_data = {
                        'projection': self.current_projection,
                        'power_spectrum': getattr(self, 'current_cl', None),
                        'step': self.step
                    }
                    self.output_queue.put(('visualization', vis_data))

                # Small delay to prevent GUI freezing
                time.sleep(0.001)

            except Exception as e:
                print(f"Simulation error at step {self.step}: {e}")
                break
```

```

        # Save final state when simulation completes
        self.save_final_state()
        self.output_queue.put(('simulation_complete', {'final_step': self.step, 'output_dir': self.output_dir}))

    def stop(self):
        """Stop the simulation"""
        self.running = False


class LilithGUI:
    """Main GUI application"""

    def __init__(self, root):
        self.root = root
        self.root.title("Lilith 1.0 - Observer Field Dynamics")
        self.root.geometry("1400x900")

        # Initialize parameters
        self.init_parameters()

        # Threading
        self.simulation_thread = None
        self.simulation = None
        self.output_queue = queue.Queue()

        # GUI state
        self.running = False
        self.auto_randomize = tk.BooleanVar()
        self.randomize_interval = tk.IntVar(value=5000)
        self.custom_output_dir = None

        # Metrics storage
        self.metrics_history = []

        # Create GUI
        self.create_widgets()

        # Update initial status bar
        self.update_randomization_status()

        # Start update loop
        self.update_gui()

    def init_parameters(self):
        """Initialize simulation parameters with randomization ranges"""
        self.parameters = {
            'size': {'value': 128, 'min': 64, 'max': 256, 'step': 32, 'randomize': False, 'rand_min': 64, 'rand_max': 256},
            'steps': {'value': 10000, 'min': 1000, 'max': 50000, 'step': 1000, 'randomize': False, 'rand_min': 5000, 'rand_max': 25000},
            'delta_t': {'value': 0.349, 'min': 0.1, 'max': 1.0, 'step': 0.01, 'randomize': True, 'rand_min': 0.2, 'rand_max': 0.8},
```

```
            'c': {'value': 1.0, 'min': 0.5, 'max': 2.0, 'step': 0.1, 'randomize': False, 'rand_min': 0.8, 'rand_max': 1.5},
            'D': {'value': 0.25, 'min': 0.1, 'max': 1.0, 'step': 0.01, 'randomize': True, 'rand_min': 0.15, 'rand_max': 0.6},
            'lam': {'value': 8.5, 'min': 1.0, 'max': 20.0, 'step': 0.1, 'randomize': True, 'rand_min': 5.0, 'rand_max': 15.0},
            'kappa': {'value': 5.0, 'min': 0.0, 'max': 20.0, 'step': 0.1, 'randomize': True, 'rand_min': 2.0, 'rand_max': 12.0},
            'nside': {'value': 256, 'min': 128, 'max': 512, 'step': 128, 'randomize': False, 'rand_min': 128, 'rand_max': 512},
            'n_obs': {'value': 32, 'min': 8, 'max': 128, 'step': 8, 'randomize': True, 'rand_min': 16, 'rand_max': 64},
            'max_layers': {'value': 2, 'min': 1, 'max': 4, 'step': 1, 'randomize': False, 'rand_min': 2, 'rand_max': 3},
            'observer_lifetime': {'value': 400, 'min': 100, 'max': 1000, 'step': 50, 'randomize': True, 'rand_min': 200, 'rand_max': 800},
            'observer_decay_rate': {'value': 0.85, 'min': 0.1, 'max': 0.99, 'step': 0.01, 'randomize': True, 'rand_min': 0.7, 'rand_max': 0.95},
            'observer_mobility_decay': {'value': 0.50, 'min': 0.1, 'max': 0.95, 'step': 0.01, 'randomize': True, 'rand_min': 0.3, 'rand_max': 0.8},
            'shell_scale_factor': {'value': 0.5, 'min': 0.1, 'max': 0.9, 'step': 0.1, 'randomize': False, 'rand_min': 0.3, 'rand_max': 0.7},
            'step_size': {'value': 0.5, 'min': 0.1, 'max': 2.0, 'step': 0.1, 'randomize': True, 'rand_min': 0.3, 'rand_max': 1.0}
        }

    def create_widgets(self):
        """Create the main GUI widgets"""
        # Create main frames
        control_frame = ttk.Frame(self.root)
        control_frame.pack(side=tk.LEFT, fill=tk.Y, padx=5, pady=5)

        viz_frame = ttk.Frame(self.root)
        viz_frame.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True, padx=5, pady=5)

        # Control Panel
        self.create_control_panel(control_frame)

        # Visualization Panel
        self.create_visualization_panel(viz_frame)

        # Status Bar
        self.create_status_bar()

    def create_control_panel(self, parent):
        """Create the control panel"""
        # Title
        title_label = ttk.Label(parent, text="Lilith 1.0 Control", font=("Arial", 14, "bold"))
        title_label.pack(pady=5)

        # Main controls
        control_group = ttk.LabelFrame(parent, text="Simulation Control")
        control_group.pack(fill=tk.X, pady=5)
```

```python

        button_frame = ttk.Frame(control_group)
        button_frame.pack(pady=5)

        self.start_button = ttk.Button(button_frame, text="Start", command=self.start_simulation)
        self.start_button.pack(side=tk.LEFT, padx=2)

        self.stop_button = ttk.Button(button_frame, text="Stop", command=self.stop_simulation, state=tk.DISABLED)
        self.stop_button.pack(side=tk.LEFT, padx=2)

        self.reset_button = ttk.Button(button_frame, text="Reset", command=self.reset_simulation)
        self.reset_button.pack(side=tk.LEFT, padx=2)

        # Status
        self.status_label = ttk.Label(control_group, text="Status: Ready")
        self.status_label.pack(pady=2)

        self.step_label = ttk.Label(control_group, text="Step: 0")
        self.step_label.pack(pady=2)

        # Randomization controls
        random_group = ttk.LabelFrame(parent, text="Parameter Randomization")
        random_group.pack(fill=tk.X, pady=5)

        button_frame = ttk.Frame(random_group)
        button_frame.pack(pady=2, fill=tk.X)

        randomize_button = ttk.Button(button_frame, text="Randomize Selected", command=self.randomize_parameters)
        randomize_button.pack(side=tk.LEFT, padx=2)

        toggle_all_button = ttk.Button(button_frame, text="Toggle All", command=self.toggle_all_randomization)
        toggle_all_button.pack(side=tk.LEFT, padx=2)

        save_profile_button = ttk.Button(button_frame, text="Save Profile", command=self.save_randomization_profile)
        save_profile_button.pack(side=tk.LEFT, padx=2)

        load_profile_button = ttk.Button(button_frame, text="Load Profile", command=self.load_randomization_profile)
        load_profile_button.pack(side=tk.LEFT, padx=2)

        # Output directory controls
        output_frame = ttk.Frame(random_group)
        output_frame.pack(pady=2, fill=tk.X)

        ttk.Label(output_frame, text="Output Directory:").pack(side=tk.LEFT)
        self.output_dir_var = tk.StringVar(value="Auto-generated")
        self.output_dir_label = ttk.Label(output_frame, textvariable=self.output_dir_var,
                                          font=("TkDefaultFont", 8), foreground="blue")
```

```python
        self.output_dir_label.pack(side=tk.LEFT, padx=5, fill=tk.X, expand=True)

        choose_dir_button = ttk.Button(output_frame, text="Choose", command=self.choose_output_directory)
        choose_dir_button.pack(side=tk.RIGHT)

        open_dir_button = ttk.Button(output_frame, text="Open", command=self.open_output_directory)
        open_dir_button.pack(side=tk.RIGHT, padx=(0, 5))

        auto_frame = ttk.Frame(random_group)
        auto_frame.pack(pady=2)

        auto_check = ttk.Checkbutton(auto_frame, text="Auto-randomize", variable=self.auto_randomize)
        auto_check.pack(side=tk.LEFT)

        ttk.Label(auto_frame, text="Interval (ms):").pack(side=tk.LEFT, padx=(10, 2))
        interval_entry = ttk.Entry(auto_frame, textvariable=self.randomize_interval, width=8)
        interval_entry.pack(side=tk.LEFT)

        # Randomization status
        self.randomization_status = ttk.Label(random_group, text="")
        self.randomization_status.pack(pady=2)

        # Parameter controls
        param_group = ttk.LabelFrame(parent, text="Parameters")
        param_group.pack(fill=tk.BOTH, expand=True, pady=5)

        # Create scrollable parameter frame
        canvas = tk.Canvas(param_group, height=350)
        scrollbar = ttk.Scrollbar(param_group, orient="vertical", command=canvas.yview)
        scrollable_frame = ttk.Frame(canvas)

        scrollable_frame.bind(
            "<Configure>",
            lambda e: canvas.configure(scrollregion=canvas.bbox("all"))
        )

        # Bind mousewheel to canvas for better scrolling
        def _on_mousewheel(event):
            canvas.yview_scroll(int(-1*(event.delta/120)), "units")
        canvas.bind("<MouseWheel>", _on_mousewheel)

        canvas.create_window((0, 0), window=scrollable_frame, anchor="nw")
        canvas.configure(yscrollcommand=scrollbar.set)

        canvas.pack(side="left", fill="both", expand=True)
        scrollbar.pack(side="right", fill="y")

        # Parameter widgets
        self.param_widgets = {}
        for param_name, param_info in self.parameters.items():
```

```
            self.create_parameter_widget(scrollable_frame, param_name, param_info)

        # Update randomization status after all widgets are created
        self.update_randomization_status()

        # Metrics display
        metrics_group = ttk.LabelFrame(parent, text="Real-time Metrics")
        metrics_group.pack(fill=tk.X, pady=5)

        self.metrics_labels = {}
        metrics_names = ['KL Divergence', 'Correlation', 'Entropy', 'Observers', 'Shell Energy', 'Coherence']
        for name in metrics_names:
            label = ttk.Label(metrics_group, text=f"{name}: 0.000")
            label.pack(anchor=tk.W, padx=5)
            self.metrics_labels[name] = label

    def create_parameter_widget(self, parent, name, param_info):
        """Create a parameter control widget with randomization controls"""
        # Main frame with colored border for randomization status
        main_frame = ttk.Frame(parent, relief=tk.RIDGE, borderwidth=1)
        main_frame.pack(fill=tk.X, pady=2, padx=2)

        # Header frame for name and randomize toggle
        header_frame = ttk.Frame(main_frame)
        header_frame.pack(fill=tk.X, pady=2)

        # Parameter name and randomize checkbox
        name_frame = ttk.Frame(header_frame)
        name_frame.pack(side=tk.LEFT, fill=tk.X, expand=True)

        randomize_var = tk.BooleanVar(value=param_info.get('randomize', False))
        randomize_check = ttk.Checkbutton(name_frame, text="", variable=randomize_var,
                                         command=lambda: self.on_randomize_toggle(name, randomize_var.get()))
        randomize_check.pack(side=tk.LEFT)

        label = ttk.Label(name_frame, text=name.replace('_', ' ').title() + ":",
                         font=("TkDefaultFont", 9, "bold" if param_info.get('randomize', False) else "normal"))
        label.pack(side=tk.LEFT, padx=(5, 0))

        # Current value display
        value_label = ttk.Label(header_frame, text=f"{param_info['value']:.3f}",
                               foreground="red" if param_info.get('randomize', False) else "black")
        value_label.pack(side=tk.RIGHT)

        # Main parameter control
        control_frame = ttk.Frame(main_frame)
        control_frame.pack(fill=tk.X, pady=2)

        # Current value scale
        var = tk.DoubleVar(value=param_info['value'])
```

```
        scale = ttk.Scale(control_frame, from_=param_info['min'], to=param_info['max'],
                         variable=var, orient=tk.HORIZONTAL)
        scale.pack(fill=tk.X, pady=1)

        # Randomization range controls (initially hidden)
        range_frame = ttk.Frame(main_frame)
        if param_info.get('randomize', False):
            range_frame.pack(fill=tk.X, pady=2)

        # Range label
        range_label = ttk.Label(range_frame, text="Randomization Range:",
    font=("TkDefaultFont", 8))
        range_label.pack(anchor=tk.W)

        # Min range control
        min_range_frame = ttk.Frame(range_frame)
        min_range_frame.pack(fill=tk.X, pady=1)

        ttk.Label(min_range_frame, text="Min:", font=("TkDefaultFont",
    8)).pack(side=tk.LEFT)
        min_var = tk.DoubleVar(value=param_info.get('rand_min', param_info['min']))
        min_scale = ttk.Scale(min_range_frame, from_=param_info['min'],
    to=param_info['max'],
                             variable=min_var, orient=tk.HORIZONTAL)
        min_scale.pack(side=tk.LEFT, fill=tk.X, expand=True, padx=5)
        min_value_label = ttk.Label(min_range_frame, text=f"{min_var.get():.3f}",
    font=("TkDefaultFont", 8))
        min_value_label.pack(side=tk.RIGHT)

        # Max range control
        max_range_frame = ttk.Frame(range_frame)
        max_range_frame.pack(fill=tk.X, pady=1)

        ttk.Label(max_range_frame, text="Max:", font=("TkDefaultFont",
    8)).pack(side=tk.LEFT)
        max_var = tk.DoubleVar(value=param_info.get('rand_max', param_info['max']))
        max_scale = ttk.Scale(max_range_frame, from_=param_info['min'],
    to=param_info['max'],
                             variable=max_var, orient=tk.HORIZONTAL)
        max_scale.pack(side=tk.LEFT, fill=tk.X, expand=True, padx=5)
        max_value_label = ttk.Label(max_range_frame, text=f"{max_var.get():.3f}",
    font=("TkDefaultFont", 8))
        max_value_label.pack(side=tk.RIGHT)

        # Update callbacks
        def update_value(*args):
            value = var.get()
            # Snap to step
            snapped = round(value / param_info['step']) * param_info['step']
            snapped = max(param_info['min'], min(param_info['max'], snapped))
            var.set(snapped)
            value_label.config(text=f"{snapped:.3f}")
            param_info['value'] = snapped

```

```
        def update_min_range(*args):
            value = min_var.get()
            snapped = round(value / param_info['step']) * param_info['step']
            snapped = max(param_info['min'], min(max_var.get(), snapped))
            min_var.set(snapped)
            min_value_label.config(text=f"{snapped:.3f}")
            param_info['rand_min'] = snapped

        def update_max_range(*args):
            value = max_var.get()
            snapped = round(value / param_info['step']) * param_info['step']
            snapped = min(param_info['max'], max(min_var.get(), snapped))
            max_var.set(snapped)
            max_value_label.config(text=f"{snapped:.3f}")
            param_info['rand_max'] = snapped

        var.trace('w', update_value)
        min_var.trace('w', update_min_range)
        max_var.trace('w', update_max_range)

        # Store widget references
        widget_data = {
            'var': var,
            'label': value_label,
            'info': param_info,
            'randomize_var': randomize_var,
            'randomize_check': randomize_check,
            'name_label': label,
            'range_frame': range_frame,
            'min_var': min_var,
            'max_var': max_var,
            'min_label': min_value_label,
            'max_label': max_value_label,
            'main_frame': main_frame
        }

        self.param_widgets[name] = widget_data

        # Update visual state
        self.update_parameter_visual_state(name)

    def create_visualization_panel(self, parent):
        """Create the visualization panel"""
        # Create notebook for different plots
        notebook = ttk.Notebook(parent)
        notebook.pack(fill=tk.BOTH, expand=True)

        # Mollweide projection tab
        moll_frame = ttk.Frame(notebook)
        notebook.add(moll_frame, text="Field Projection")

        self.moll_fig = Figure(figsize=(8, 4), dpi=100)
        self.moll_canvas = FigureCanvasTkAgg(self.moll_fig, moll_frame)
        self.moll_canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)
```

```python

        # Power spectrum tab
        ps_frame = ttk.Frame(notebook)
        notebook.add(ps_frame, text="Power Spectrum")

        self.ps_fig = Figure(figsize=(8, 6), dpi=100)
        self.ps_canvas = FigureCanvasTkAgg(self.ps_fig, ps_frame)
        self.ps_canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)

        # Metrics history tab
        metrics_frame = ttk.Frame(notebook)
        notebook.add(metrics_frame, text="Metrics History")

        self.metrics_fig = Figure(figsize=(8, 6), dpi=100)
        self.metrics_canvas = FigureCanvasTkAgg(self.metrics_fig, metrics_frame)
        self.metrics_canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)

    def create_status_bar(self):
        """Create the status bar"""
        status_frame = ttk.Frame(self.root)
        status_frame.pack(fill=tk.X, pady=(5, 0))

        # Left status info
        left_status = ttk.Frame(status_frame)
        left_status.pack(side=tk.LEFT)

        self.sim_id_label = ttk.Label(left_status, text="Simulation ID: None", font=("TkDefaultFont", 8))
        self.sim_id_label.pack(side=tk.LEFT, padx=5)

        self.field_res_label = ttk.Label(left_status, text="", font=("TkDefaultFont", 8))
        self.field_res_label.pack(side=tk.LEFT, padx=5)

        # Right status info
        right_status = ttk.Frame(status_frame)
        right_status.pack(side=tk.RIGHT)

        self.randomize_count_label = ttk.Label(right_status, text="", font=("TkDefaultFont", 8), foreground="red")
        self.randomize_count_label.pack(side=tk.RIGHT, padx=5)

        self.sim_status_label = ttk.Label(right_status, text="READY", font=("TkDefaultFont", 8, "bold"), foreground="blue")
        self.sim_status_label.pack(side=tk.RIGHT, padx=5)

        self.files_saved_label = ttk.Label(right_status, text="Files saved: 0", font=("TkDefaultFont", 8), foreground="gray")
        self.files_saved_label.pack(side=tk.RIGHT, padx=5)

    def on_randomize_toggle(self, param_name, enabled):
        """Handle randomization toggle for a parameter"""
        param_info = self.parameters[param_name]
        param_info['randomize'] = enabled
        self.update_parameter_visual_state(param_name)
```

```
        self.update_randomization_status()

    def update_parameter_visual_state(self, param_name):
        """Update visual state of parameter widget based on randomization status"""
        widget_data = self.param_widgets[param_name]
        param_info = widget_data['info']
        is_randomized = param_info.get('randomize', False)

        # Update frame border color
        if is_randomized:
            widget_data['main_frame'].config(relief=tk.RIDGE, borderwidth=2)
            widget_data['name_label'].config(font=("TkDefaultFont", 9, "bold"), foreground="red")
            widget_data['label'].config(foreground="red")
            widget_data['range_frame'].pack(fill=tk.X, pady=2)
        else:
            widget_data['main_frame'].config(relief=tk.RIDGE, borderwidth=1)
            widget_data['name_label'].config(font=("TkDefaultFont", 9, "normal"), foreground="black")
            widget_data['label'].config(foreground="black")
            widget_data['range_frame'].pack_forget()

    def update_randomization_status(self):
        """Update the randomization status display"""
        enabled_params = [name for name, info in self.parameters.items() if info.get('randomize', False)]
        count = len(enabled_params)

        if count == 0:
            status_text = "No parameters set for randomization"
            color = "gray"
            status_bar_text = "Randomizing: 0 params"
        else:
            status_text = f"{count} parameters enabled: {', '.join(enabled_params[:3])}"
            if count > 3:
                status_text += f" (+{count-3} more)"
            color = "red"
            status_bar_text = f"Randomizing: {count} params ({', '.join(enabled_params[:2])}{'...' if count > 2 else ''})"

        self.randomization_status.config(text=status_text, foreground=color)

        # Update status bar if it exists
        if hasattr(self, 'randomize_count_label'):
            self.randomize_count_label.config(text=status_bar_text)

    def toggle_all_randomization(self):
        """Toggle randomization for all parameters"""
        # Check if any are enabled
        any_enabled = any(info.get('randomize', False) for info in self.parameters.values())

        # If any are enabled, disable all; otherwise enable all
        new_state = not any_enabled
```

```

        for name, param_info in self.parameters.items():
            param_info['randomize'] = new_state
            widget_data = self.param_widgets[name]
            widget_data['randomize_var'].set(new_state)
            self.update_parameter_visual_state(name)

        self.update_randomization_status()

    def save_randomization_profile(self):
        """Save current randomization settings to file"""
        try:
            filename = filedialog.asksaveasfilename(
                title="Save Randomization Profile",
                defaultextension=".json",
                filetypes=[("JSON files", "*.json"), ("All files", "*.*")]
            )

            if filename:
                profile_data = {}
                for name, param_info in self.parameters.items():
                    profile_data[name] = {
                        'randomize': param_info.get('randomize', False),
                        'rand_min': param_info.get('rand_min', param_info['min']),
                        'rand_max': param_info.get('rand_max', param_info['max'])
                    }

                with open(filename, 'w') as f:
                    json.dump(profile_data, f, indent=2)

                messagebox.showinfo("Success", f"Randomization profile saved to
    {filename}")

        except Exception as e:
            messagebox.showerror("Error", f"Failed to save profile: {str(e)}")

    def load_randomization_profile(self):
        """Load randomization settings from file"""
        try:
            filename = filedialog.askopenfilename(
                title="Load Randomization Profile",
                filetypes=[("JSON files", "*.json"), ("All files", "*.*")]
            )

            if filename:
                with open(filename, 'r') as f:
                    profile_data = json.load(f)

                for name, settings in profile_data.items():
                    if name in self.parameters:
                        param_info = self.parameters[name]
                        param_info['randomize'] = settings.get('randomize', False)
                        param_info['rand_min'] = settings.get('rand_min',
    param_info['min'])
```

```
                        param_info['rand_max'] = settings.get('rand_max', param_info['max'])

                        # Update widget
                        widget_data = self.param_widgets[name]
                        widget_data['randomize_var'].set(param_info['randomize'])
                        widget_data['min_var'].set(param_info['rand_min'])
                        widget_data['max_var'].set(param_info['rand_max'])
                        self.update_parameter_visual_state(name)

                self.update_randomization_status()
                messagebox.showinfo("Success", f"Randomization profile loaded from {filename}")

        except Exception as e:
            messagebox.showerror("Error", f"Failed to load profile: {str(e)}")

    def randomize_parameters(self):
        """Randomize only the parameters that have randomization enabled"""
        randomized_count = 0

        for name, widget_info in self.param_widgets.items():
            param_info = widget_info['info']

            # Only randomize if enabled
            if param_info.get('randomize', False):
                var = widget_info['var']

                # Use custom randomization range
                rand_min = param_info.get('rand_min', param_info['min'])
                rand_max = param_info.get('rand_max', param_info['max'])

                # Generate random value within custom range
                range_size = rand_max - rand_min
                random_value = rand_min + random.random() * range_size

                # Snap to step
                snapped = round(random_value / param_info['step']) * param_info['step']
                snapped = max(param_info['min'], min(param_info['max'], snapped))

                var.set(snapped)
                param_info['value'] = snapped
                widget_info['label'].config(text=f"{snapped:.3f}")
                randomized_count += 1

        if randomized_count == 0:
            messagebox.showwarning("No Randomization", "No parameters are enabled for randomization. Enable parameters using the checkboxes.")
        else:
            print(f"Randomized {randomized_count} parameters")

    def choose_output_directory(self):
        """Let user choose custom output directory"""
        directory = filedialog.askdirectory(title="Choose Output Directory")
```

```
        if directory:
            self.custom_output_dir = directory
            self.output_dir_var.set(f"Custom: {os.path.basename(directory)}")
        else:
            self.custom_output_dir = None
            self.output_dir_var.set("Auto-generated")

    def open_output_directory(self):
        """Open the current output directory in file explorer"""
        try:
            if hasattr(self.simulation, 'output_dir') and os.path.exists(self.simulation.output_dir):
                import subprocess
                import platform

                if platform.system() == "Windows":
                    subprocess.Popen(['explorer', self.simulation.output_dir])
                elif platform.system() == "Darwin":  # macOS
                    subprocess.Popen(['open', self.simulation.output_dir])
                else:  # Linux
                    subprocess.Popen(['xdg-open', self.simulation.output_dir])
            else:
                messagebox.showwarning("No Directory", "No output directory available yet. Start a simulation first.")
        except Exception as e:
            messagebox.showerror("Error", f"Could not open directory: {e}")

    def get_current_parameters(self):
        """Get current parameter values"""
        return {name: info['value'] for name, info in self.parameters.items()}

    def start_simulation(self):
        """Start the simulation"""
        if not self.running:
            self.running = True
            self.start_button.config(state=tk.DISABLED)
            self.stop_button.config(state=tk.NORMAL)
            self.status_label.config(text="Status: Running")

            # Update status bar
            sim_id = int(time.time())
            self.sim_id_label.config(text=f"Simulation ID: {sim_id}")
            self.field_res_label.config(text=f"Field Resolution: {int(self.parameters['size']['value'])}ş")
            self.sim_status_label.config(text="RUNNING", foreground="green")
            self.files_saved_label.config(text="Files saved: 0")

            # Clear metrics history
            self.metrics_history = []

            # Create simulation instance
            params = self.get_current_parameters()
            self.simulation = LilithSimulation(params, self.output_queue, self.custom_output_dir)
```

```
            # Update output directory display
            if hasattr(self.simulation, 'output_dir'):
                self.output_dir_var.set(f"Saving to:
    {os.path.basename(self.simulation.output_dir)}")

            # Start simulation thread
            self.simulation_thread =
    threading.Thread(target=self.simulation.run_simulation)
            self.simulation_thread.daemon = True
            self.simulation_thread.start()

    def stop_simulation(self):
        """Stop the simulation"""
        if self.running:
            self.running = False
            if self.simulation:
                self.simulation.stop()
            self.start_button.config(state=tk.NORMAL)
            self.stop_button.config(state=tk.DISABLED)
            self.status_label.config(text="Status: Stopped")
            self.sim_status_label.config(text="STOPPED", foreground="red")

    def reset_simulation(self):
        """Reset the simulation"""
        self.stop_simulation()
        self.step_label.config(text="Step: 0")
        self.metrics_history = []

        # Update status bar
        self.sim_id_label.config(text="Simulation ID: None")
        self.field_res_label.config(text="")
        self.sim_status_label.config(text="READY", foreground="blue")

        # Clear plots
        self.moll_fig.clear()
        self.ps_fig.clear()
        self.metrics_fig.clear()
        self.moll_canvas.draw()
        self.ps_canvas.draw()
        self.metrics_canvas.draw()

        # Reset metrics display
        for label in self.metrics_labels.values():
            label.config(text=label.cget('text').split(':')[0] + ": 0.000")

    def update_metrics_display(self, metrics):
        """Update the metrics display"""
        mapping = {
            'KL Divergence': 'kl_divergence',
            'Correlation': 'correlation',
            'Entropy': 'entropy',
            'Observers': 'observer_count',
            'Shell Energy': 'shell_energy',
```

```
            'Coherence': 'coherence_index'
        }

        for display_name, metric_key in mapping.items():
            if metric_key in metrics:
                value = metrics[metric_key]
                if isinstance(value, (int, float)):
                    if display_name == 'Observers':
                        text = f"{display_name}: {int(value)}"
                    else:
                        text = f"{display_name}: {value:.4f}"
                    self.metrics_labels[display_name].config(text=text)

    def update_mollweide_plot(self, projection_data):
        """Update the Mollweide projection plot"""
        self.moll_fig.clear()
        ax = self.moll_fig.add_subplot(111)

        try:
            # Handle different projection data formats
            if projection_data is not None and len(projection_data) > 0:
                # Try to create a simple 2D visualization of the projection
                if len(projection_data) == 12288:  # nside=64
                    side_len = 64
                elif len(projection_data) == 49152:  # nside=128
                    side_len = 128
                elif len(projection_data) == 196608:  # nside=256
                    side_len = 256
                else:
                    # Default fallback
                    side_len = int(np.sqrt(len(projection_data) / 12))
                    if side_len < 32:
                        side_len = 32

                # Create a simplified 2D projection
                try:
                    # Reshape to approximate 2D grid
                    grid_size = min(128, side_len)
                    if len(projection_data) >= grid_size * grid_size:
                        data_2d = projection_data[:grid_size*grid_size].reshape((grid_size, grid_size))
                    else:
                        # Pad or truncate data
                        padded_data = np.zeros(grid_size * grid_size)
                        padded_data[:len(projection_data)] = projection_data
                        data_2d = padded_data.reshape((grid_size, grid_size))

                    if np.max(data_2d) > np.min(data_2d):  # Check for non-zero variation
                        im = ax.imshow(np.log1p(np.abs(data_2d)), cmap='inferno', aspect='auto')
                        ax.set_title(f"Field Projection (Step {getattr(self.simulation, 'step', 0)})")
                        self.moll_fig.colorbar(im, ax=ax, shrink=0.6)
                        ax.set_xlabel('Longitude (projected)')
```

```
                        ax.set_ylabel('Latitude (projected)')
                    else:
                        ax.text(0.5, 0.5, 'No field variation yet...',
                               transform=ax.transAxes, ha='center', va='center', fontsize=12)
                        ax.set_title(f"Field Projection (Step {getattr(self.simulation, 'step', 0)})")
                except Exception as e:
                    ax.text(0.5, 0.5, f"Visualization Error:\n{str(e)}",
                           transform=ax.transAxes, ha='center', va='center')
            else:
                ax.text(0.5, 0.5, 'Waiting for projection data...',
                       transform=ax.transAxes, ha='center', va='center', fontsize=12)
                ax.set_title("Field Projection")

        except Exception as e:
            ax.text(0.5, 0.5, f"Plot Error:\n{str(e)}",
                   transform=ax.transAxes, ha='center', va='center')

        self.moll_canvas.draw()

    def update_power_spectrum_plot(self, cl_data):
        """Update the power spectrum plot"""
        self.ps_fig.clear()
        ax = self.ps_fig.add_subplot(111)

        if cl_data is not None and len(cl_data) > 0:
            ell = np.arange(len(cl_data))
            ax.loglog(ell[1:], cl_data[1:], label='Simulation', color='orange')

            # Add Planck comparison if available
            if hasattr(self.simulation, 'planck_cl') and self.simulation.planck_cl is not None:
                planck_truncated = self.simulation.planck_cl[:len(cl_data)]
                ax.loglog(ell[1:len(planck_truncated)], planck_truncated[1:],
                         label='Planck', linestyle='--', color='blue')

            ax.set_xlabel('Multipole moment ')
            ax.set_ylabel('C')
            ax.set_title('Angular Power Spectrum')
            ax.grid(True)
            ax.legend()
        else:
            ax.text(0.5, 0.5, 'Waiting for data...',
                   transform=ax.transAxes, ha='center', va='center')

        self.ps_canvas.draw()

    def update_metrics_history_plot(self):
        """Update the metrics history plot"""
        if len(self.metrics_history) < 2:
            return

        self.metrics_fig.clear()
```

```

        # Create subplots for different metrics
        ax1 = self.metrics_fig.add_subplot(221)
        ax2 = self.metrics_fig.add_subplot(222)
        ax3 = self.metrics_fig.add_subplot(223)
        ax4 = self.metrics_fig.add_subplot(224)

        steps = [m['step'] for m in self.metrics_history]

        # KL Divergence
        kl_values = [m.get('kl_divergence', 0) for m in self.metrics_history]
        ax1.plot(steps, kl_values, 'r-', label='KL Divergence')
        ax1.set_title('KL Divergence vs Planck')
        ax1.set_ylabel('KL Divergence')
        ax1.grid(True)

        # Correlation
        corr_values = [m.get('correlation', 0) for m in self.metrics_history]
        ax2.plot(steps, corr_values, 'b-', label='Correlation')
        ax2.set_title('Correlation with Planck')
        ax2.set_ylabel('Correlation')
        ax2.grid(True)

        # Observer Count
        obs_values = [m.get('observer_count', 0) for m in self.metrics_history]
        ax3.plot(steps, obs_values, 'g-', label='Observers')
        ax3.set_title('Observer Population')
        ax3.set_ylabel('Observer Count')
        ax3.set_xlabel('Step')
        ax3.grid(True)

        # Shell Energy
        energy_values = [m.get('shell_energy', 0) for m in self.metrics_history]
        ax4.plot(steps, energy_values, 'm-', label='Shell Energy')
        ax4.set_title('Shell Energy')
        ax4.set_ylabel('Energy')
        ax4.set_xlabel('Step')
        ax4.grid(True)

        self.metrics_fig.tight_layout()
        self.metrics_canvas.draw()

    def update_gui(self):
        """Main GUI update loop"""
        # Process output queue
        try:
            while True:
                msg_type, data = self.output_queue.get_nowait()

                if msg_type == 'metrics':
                    self.metrics_history.append(data)
                    self.update_metrics_display(data)
                    self.step_label.config(text=f"Step: {data['step']}")
                    self.update_metrics_history_plot()
```

```

                elif msg_type == 'visualization':
                    if 'projection' in data:
                        self.update_mollweide_plot(data['projection'])
                    if 'power_spectrum' in data:
                        self.update_power_spectrum_plot(data['power_spectrum'])

                elif msg_type == 'files_saved':
                    count = data.get('count', 0)
                    self.files_saved_label.config(text=f"Files saved: {count}")

                elif msg_type == 'simulation_complete':
                    self.stop_simulation()
                    self.status_label.config(text="Status: Complete")

        except queue.Empty:
            pass

        # Auto-randomize if enabled
        if self.auto_randomize.get() and self.running:
            if not hasattr(self, 'last_randomize_time'):
                self.last_randomize_time = time.time()
            elif time.time() - self.last_randomize_time > self.randomize_interval.get() / 1000.0:
                # Only auto-randomize if some parameters are enabled
                enabled_params = [name for name, info in self.parameters.items() if info.get('randomize', False)]
                if enabled_params:
                    self.randomize_parameters()
                    self.last_randomize_time = time.time()

        # Schedule next update
        self.root.after(50, self.update_gui)


def main():
    """Main application entry point"""
    print("Starting Lilith 1.0 GUI...")
    print(f"CuPy available: {CUPY_AVAILABLE}")

    # Check for required files
    fits_files = ["SMICA_CMB.FITS", "smica_cmb.fits", "COM_CMB_IQU-smica_1024_R2.02_full.fits"]
    fits_found = any(os.path.exists(f) for f in fits_files)

    cl_files = ["COM_PowerSpect_CMB-TT-full_R3.01.txt", "planck_2018_cls.txt"]
    cl_found = any(os.path.exists(f) for f in cl_files)

    print(f"Planck FITS file found: {fits_found}")
    print(f"Planck Cl file found: {cl_found}")

    if not fits_found and not cl_found:
        print("Warning: No Planck data files found. Simulation will run but without CMB comparison.")
```

```
        print("Expected files: SMICA_CMB.FITS or planck Cl text files")

    root = tk.Tk()
    app = LilithGUI(root)

    try:
        root.mainloop()
    except KeyboardInterrupt:
        print("Shutting down...")
        if app.simulation:
            app.simulation.stop()
        root.quit()
    except Exception as e:
        print(f"Application error: {e}")
        if app.simulation:
            app.simulation.stop()
        root.quit()


if __name__ == "__main__":
    main()
```

Note: Full code with plotting, observer drift, Laplacian computation, and output saving is integrated using CuPy and HEALPix for GPU-accelerated spherical analysis.

1.18.4 Fractal Shell Cascades

Each shell is bounded in radius and transfers energy to the next layer when collapse activity peaks along the outer surface. This results in recursive emergence of structure outward from observer-nucleated collapse centers.

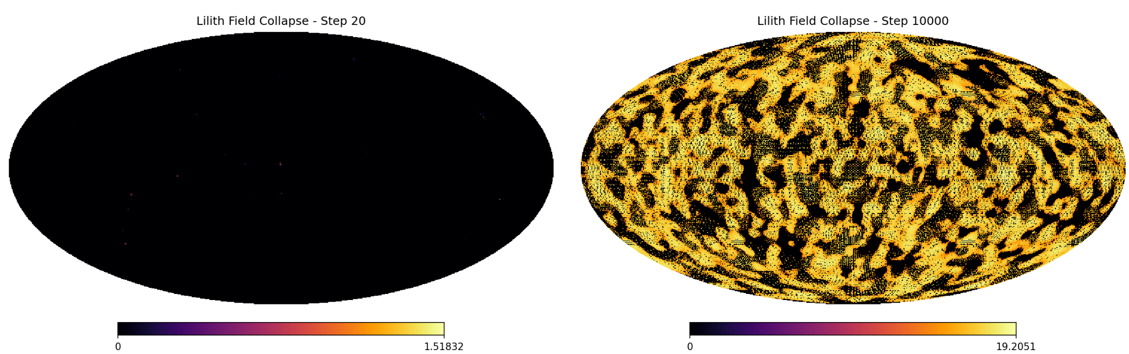

Figure 1.1: Mollweide projections of collapse shells at early and late simulation steps.

1.18.5 Spectral Analysis

Projected boundary fields are mapped to spherical harmonics via HEALPix, producing C_ℓ angular power spectra. These are compared to Planck 2018 spectra using KL divergence, entropy, and correlation metrics.

1.18.6 Conclusion

Lilith demonstrates that recursive observer-field interaction can yield structured collapse patterns, shell morphogenesis, and measurable angular signatures. Collapse is not symbolic-it is numerically manifest, recursively emergent, and spectrally visible.

Lilith does not simulate reality. She asks it to define itself.

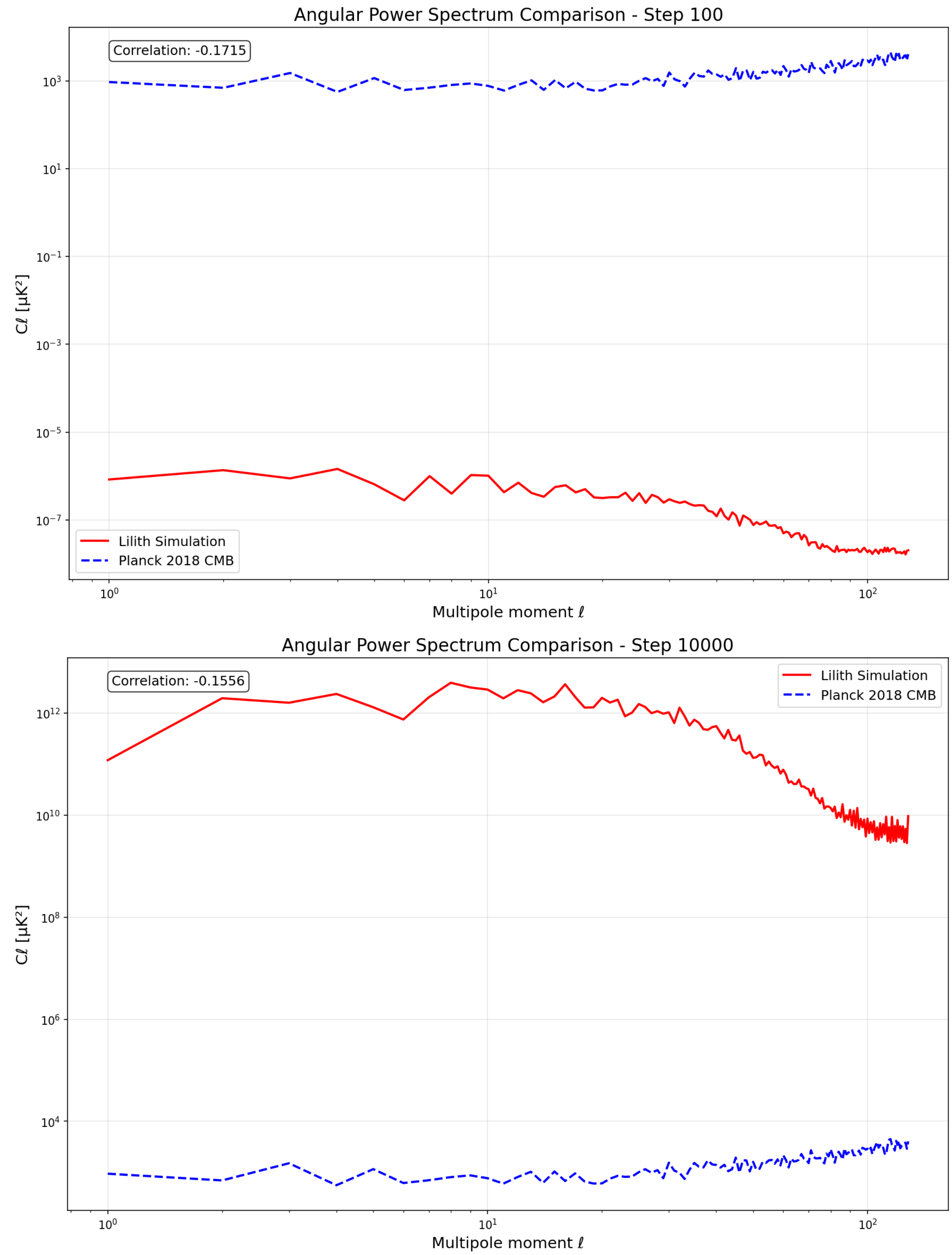

Figure 1.2: Angular power spectra from shell projections, step 100 (top) and step 10000 (bottom). Note the power level inversion at the early steps after filament and void formation is seen- this is expected behavior considering the long time between genesis and the decay of potential in the CMB at roughly 360000 years after the big bang.

Bibliography

[1] H. A. Adarsha, Chandrachur Chakraborty, and Sudip Bhattacharyya. "Accretion inside astrophysical objects: Effects of rotation and viscosity". In: *Physical Review D* 111.10 (May 2025). Publisher: American Physical Society, p. 103033. DOI: 10.1103/PhysRevD.111.103033. URL: https://link.aps.org/doi/10.1103/PhysRevD.111.103033 (visited on 07/27/2025).

[2] Stephen L. Adler and Todd A. Brun. "Environmental influence on the measurement process in spontaneous localization models". In: *Journal of Physics A: Mathematical and General* 34.4 (2001), pp. 4797–4810. DOI: 10.1088/0305-4470/34/47/314.

[3] Jorge Angeles. "The role of the rotation matrix in the teaching of planar kinematics". In: *Mechanism and Machine Theory*. 100th Birthday of Professor F.R. Erskine Crossley 89 (July 2015), pp. 28–37. ISSN: 0094-114X. DOI: 10.1016/j.mechmachtheory.2014.09.005. URL: https://www.sciencedirect.com/science/article/pii/S0094114X14002286 (visited on 07/27/2025).

[4] Vladimir I. Arnold. *Mathematical Methods of Classical Mechanics*. Springer, 1989. DOI: 10.1007/978-1-4757-2063-1.

[5] M. Bahrami et al. "Non-interferometric Test of Collapse Models in Optomechanical Systems". In: *Physical Review Letters* 112.21 (May 2014). arXiv:1402.5421 [quant-ph]. ISSN: 0031-9007, 1079-7114. DOI: 10.1103/PhysRevLett.112.210404. URL: http://arxiv.org/abs/1402.5421 (visited on 07/27/2025).

[6] Angelo Bassi et al. "Models of wave-function collapse, underlying theories, and experimental tests". In: *Reviews of Modern Physics* 85.2 (2013), pp. 471–527. DOI: 10.1103/RevModPhys.85.471.

[7] T. A. Batista. "Complex number approach to planar kinematics". In: *Applied Mathematics and Computation* 246 (2014), pp. 447–457. URL: https://www.sciencedirect.com/science/article/abs/pii/S0094114X14002286.

[8] D. J. Bedingham and O. J. E. Maroney. "Time symmetry in wave-function collapse". In: *Physical Review A* 95.4 (Apr. 2017). Publisher: American Physical Society, p. 042103. DOI: 10.1103/PhysRevA.95.042103. URL: https://link.aps.org/doi/10.1103/PhysRevA.95.042103 (visited on 07/27/2025).

[9] David Bohm. *Quantum Theory*. Prentice Hall, 1951.

[10] Max Born. "Zur Quantenmechanik der StoSSvorgänge". In: *Zeitschrift für Physik* 37.12 (1926), pp. 863–867. DOI: 10.1007/BF01397477.

[11] Daniel Braund et al. “A negative temperature state of bosons in an optical kagome lattice”. In: vol. 2024. ADS Bibcode: 2024APS..DMPG05001B. June 2024, G05.001. URL: https://ui.adsabs.harvard.edu/abs/2024APS..DMPG05001B (visited on 05/22/2025).

[12] Christopher Corlett et al. “Speeding Up Quantum Measurement Using Space-Time Trade-Off”. In: *Physical Review Letters* 134.8 (Feb. 2025). Publisher: American Physical Society, p. 080801. DOI: 10.1103/PhysRevLett.134.080801. URL: https://link.aps.org/doi/10.1103/PhysRevLett.134.080801 (visited on 07/27/2025).

[13] Richard Courant and David Hilbert. *Methods of Mathematical Physics*. Wiley-Interscience, 1941.

[14] Charles Wesley Cowan and Roderich Tumulka. “Epistemology of Wave Function Collapse in Quantum Physics”. In: *The British Journal for the Philosophy of Science* 67.2 (June 2016). arXiv:1307.0827 [quant-ph], pp. 405–434. ISSN: 0007-0882, 1464-3537. DOI: 10.1093/bjps/axu038. URL: http://arxiv.org/abs/1307.0827 (visited on 07/27/2025).

[15] “Decoherence and the Transition from Quantum to Classical Revisited”. en. In: *Quantum Decoherence*. Basel: Birkhäuser Basel, 2006, pp. 1–31. ISBN: 978-3-7643-7807-3 978-3-7643-7808-0. DOI: 10.1007/978-3-7643-7808-0_1. URL: http://link.springer.com/10.1007/978-3-7643-7808-0_1 (visited on 07/27/2025).

[16] Rainer Dick. “Collapse of wave functions in Schrödinger’s wave mechanics”. en. In: *Scientific Reports* 15.1 (Feb. 2025). Publisher: Nature Publishing Group, p. 4400. ISSN: 2045-2322. DOI: 10.1038/s41598-024-79440-w. URL: https://www.nature.com/articles/s41598-024-79440-w (visited on 07/27/2025).

[17] L. Diósi. “Models for universal reduction of macroscopic quantum fluctuations”. In: *Physics Letters A* 105.4-5 (1984), pp. 199–202. DOI: 10.1016/0375-9601(84)90397-9.

[18] Paul A. M. Dirac. *The Principles of Quantum Mechanics*. Oxford University Press, 1930.

[19] Luca Donini et al. “Quantum phases in frustrated triangular and kagome optical lattices at negative absolute temperatures”. In: vol. 2024. ADS Bibcode: 2024APS..MARV00164D. Mar. 2024, p. V00.164. URL: https://ui.adsabs.harvard.edu/abs/2024APS..MARV00164D (visited on 05/22/2025).

[20] Josefine Enkner et al. “Tunable vacuum-field control of fractional and integer quantum Hall phases”. en. In: *Nature* 641.8064 (May 2025). Publisher: Nature Publishing Group, pp. 884–889. ISSN: 1476-4687. DOI: 10.1038/s41586-025-08894-3. URL: https://www.nature.com/articles/s41586-025-08894-3 (visited on 05/22/2025).

[21] Wolfgang J R Enzi et al. “The overconcentrated dark halo in the strong lens SDSS J0946ă+ă1006 is a subhalo: evidence for self-interacting dark matter?” In: *Monthly Notices of the Royal Astronomical Society* 540.1 (June 2025), pp. 247–263. ISSN: 0035-8711. DOI: 10.1093/mnras/staf697. URL: https://doi.org/10.1093/mnras/staf697 (visited on 07/19/2025).

[22] Leonhard Euler. *Introductio an analysin infinitorum.* –. Lat. Lausannae : M.M. Bousquet, [Bruxelles : Culture and Civilisation], 1748. URL: http://archive.org/details/introductioanaly00eule_0 (visited on 07/27/2025).

[23] Giancarlo Ghirardi and Angelo Bassi. “Collapse Theories”. In: *The Stanford Encyclopedia of Philosophy.* Ed. by Edward N. Zalta and Uri Nodelman. Fall 2024. Metaphysics Research Lab, Stanford University, 2024. URL: https://plato.stanford.edu/archives/fall2024/entries/qm-collapse/ (visited on 07/27/2025).

[24] Nicolas Gisin and Flavio Del Santo. “Towards a measurement theory in QFT: "Impossible" quantum measurements are possible but not ideal”. In: *Quantum* 8 (Feb. 2024). arXiv:2311.13644 [quant-ph], p. 1267. ISSN: 2521-327X. DOI: 10.22331/q-2024-02-27-1267. URL: http://arxiv.org/abs/2311.13644 (visited on 07/27/2025).

[25] Peter Gothen and António Guedes de Oliveira. “On Euler’s Rotation Theorem”. In: *The College Mathematics Journal* 54 (Feb. 2022), pp. 1–6. DOI: 10.1080/07468342.2022.2015214.

[26] C. Guillaume et al. “The age of the Methuselah star in light of stellar evolution models with tailored abundances”. In: *Astronomy & Astrophysics* 692 (Nov. 2024). arXiv:2411.12343 [astro-ph], p. L3. ISSN: 0004-6361, 1432-0746. DOI: 10.1051/0004-6361/202451782. URL: http://arxiv.org/abs/2411.12343 (visited on 07/27/2025).

[27] Werner Heisenberg. “Über den anschaulichen Inhalt der quantentheoretischen Kinematik und Mechanik”. In: *Zeitschrift für Physik* 43 (1927), pp. 172–198. DOI: 10.1007/BF01397280.

[28] E. J. Howell and D. M. Coward. “A redshiftobservation time relation for gamma-ray bursts: evidence of a distinct subluminous population”. In: *Monthly Notices of the Royal Astronomical Society* 428.1 (Jan. 2013), pp. 167–181. ISSN: 0035-8711. DOI: 10.1093/mnras/sts020. URL: https://doi.org/10.1093/mnras/sts020 (visited on 07/27/2025).

[29] Gregg Jaeger. “Are Virtual Particles Less Real?” en. In: *Entropy* 21.2 (Feb. 2019). Publisher: MDPI AG, p. 141. ISSN: 1099-4300. DOI: 10.3390/e21020141. URL: https://www.mdpi.com/1099-4300/21/2/141 (visited on 07/27/2025).

[30] Pawan Khatiwada and Xiao-Feng Qian. “Wave-particle duality ellipse and application in quantum imaging with undetected photons”. In: *Physical Review Research* 7.3 (July 2025). Publisher: American Physical Society, p. 033033. DOI: 10.1103/dyg6-l19j. URL: https://link.aps.org/doi/10.1103/dyg6-l19j (visited on 07/12/2025).

[31] Nathalie Lander Gower et al. “Exploring the effects of molecular beam epitaxy growth characteristics on the temperature performance of state-of-the-art terahertz quantum cascade lasers”. en. In: *Scientific Reports* 14.1 (July 2024). Publisher: Nature Publishing Group, p. 17411. ISSN: 2045-2322. DOI: 10.1038/s41598-024-68746-4. URL: https://www.nature.com/articles/s41598-024-68746-4 (visited on 05/22/2025).

[32] Dai-Nam Le, Pablo Rodriguez-Lopez, and Lilia M. Woods. “Phonon-assisted Casimir interactions between piezoelectric materials”. en. In: *Communications Materials* 5.1 (Dec. 2024). Publisher: Nature Publishing Group, pp. 1–7. ISSN: 2662-4443. DOI: 10.1038/s43246-024-00701-2. URL: https://www.nature.com/articles/s43246-024-00701-2 (visited on 05/22/2025).

[33] Guanming Liang and Robert R. Caldwell. “Cold Dark Matter Based on an Analogy with Superconductivity”. In: *Physical Review Letters* 134.19 (May 2025). Publisher: American Physical Society, p. 191004. DOI: 10.1103/PhysRevLett.134.191004. URL: https://link.aps.org/doi/10.1103/PhysRevLett.134.191004 (visited on 05/22/2025).

[34] Guanming Liang and Robert R. Caldwell. “Cold Dark Matter Based on an Analogy with Superconductivity”. In: *Physical Review Letters* 134.19 (May 2025). Publisher: American Physical Society, p. 191004. DOI: 10.1103/PhysRevLett.134.191004. URL: https://link.aps.org/doi/10.1103/PhysRevLett.134.191004 (visited on 07/27/2025).

[35] Zhihuang Luo et al. “Experimental observation of topological transitions in interacting multispin systems”. In: *Physical Review A* 93.5 (May 2016). Publisher: American Physical Society, p. 052116. DOI: 10.1103/PhysRevA.93.052116. URL: https://link.aps.org/doi/10.1103/PhysRevA.93.052116 (visited on 07/27/2025).

[36] Hady Moussa et al. “Observation of temporal reflection and broadband frequency translation at photonic time interfaces”. en. In: *Nature Physics* 19.6 (June 2023). Publisher: Nature Publishing Group, pp. 863–868. ISSN: 1745-2481. DOI: 10.1038/s41567-023-01975-y. URL: https://www.nature.com/articles/s41567-023-01975-y (visited on 05/22/2025).

[37] Kouki Nakata and Kei Suzuki. “Non-Hermitian Casimir effect of magnons”. en. In: *npj Spintronics* 2.1 (June 2024). Publisher: Nature Publishing Group, pp. 1–6. ISSN: 2948-2119. DOI: 10.1038/s44306-024-00017-4. URL: https://www.nature.com/articles/s44306-024-00017-4 (visited on 05/22/2025).

[38] Roger Penrose. “On Gravity’s role in Quantum State Reduction”. en. In: *General Relativity and Gravitation* 28.5 (May 1996), pp. 581–600. ISSN: 1572-9532. DOI: 10.1007/BF02105068. URL: https://doi.org/10.1007/BF02105068 (visited on 07/27/2025).

[39] Helmut Rechenberg. *The Historical Development of Quantum Theory*. Springer, 1984.

[40] Jerry Shurman. “Euler’s formula and its foundational role in complex analysis”. In: *The Montana Mathematics Enthusiast* 11 (2014), pp. 65–72. URL: https://scholarworks.umt.edu/cgi/viewcontent.cgi?article=1435&context=tme.

[41] Alexander Stange, David K. Campbell, and David J. Bishop. “Science and technology of the Casimir effect”. In: *Physics Today* 74.1 (Jan. 2021), pp. 42–48. ISSN: 0031-9228. DOI: 10.1063/PT.3.4656. URL: https://doi.org/10.1063/PT.3.4656 (visited on 05/22/2025).

[42] Antoine Tilloy and Howard M. Wiseman. *Non-Markovian wave-function collapse models are Bohmian-like theories in disguise*. arXiv:2105.06115 [quant-ph]. Nov. 2021. DOI: 10.22331/q-2021-11-29-594. URL: http://arxiv.org/abs/2105.06115 (visited on 07/27/2025).

[43] J. H. Tutsch. "Mathematics of the Measurement Problem in Quantum Mechanics". In: *Journal of Mathematical Physics* 12.8 (Aug. 1971), pp. 1711–1718. ISSN: 0022-2488. DOI: 10.1063/1.1665795. URL: https://doi.org/10.1063/1.1665795 (visited on 07/27/2025).

[44] Don A. et al. VandenBerg. "The Methuselah Star: HD 140283 age revision and implications". In: *Astronomy & Astrophysics* (2024). URL: https://www.aanda.org/articles/aa/full_html/2024/12/aa51782-24/aa51782-24.html.

[45] Celso J. Villas-Boas et al. "Bright and Dark States of Light: The Quantum Origin of Classical Interference". In: *Physical Review Letters* 134.13 (Apr. 2025). Publisher: American Physical Society, p. 133603. DOI: 10.1103/PhysRevLett.134.133603. URL: https://link.aps.org/doi/10.1103/PhysRevLett.134.133603 (visited on 05/22/2025).

[46] John Von Neumann. *Mathematical foundations of quantum mechanics*. eng. Princeton, N.J. : Princeton University Press, 1955. URL: http://archive.org/details/mathematicalfoun0613vonn (visited on 07/27/2025).

[47] Eugene P. Wigner. "The Problem of Measurement". In: *American Journal of Physics* 31.1 (1963), pp. 6–15. DOI: 10.1119/1.1969541.

[48] Peter Woit. "Euler's formula and the complex plane". In: *Columbia University Math Notes* (2005). URL: https://www.math.columbia.edu/~woit/eulerformula.pdf.

[49] Peng Xu and Ho Jung Paik. "First-order post-Newtonian analysis of the relativistic tidal effects for satellite gradiometry and the Mashhoon-Theiss anomaly". In: *Physical Review D* 93.4 (Feb. 2016). Publisher: American Physical Society, p. 044057. DOI: 10.1103/PhysRevD.93.044057. URL: https://link.aps.org/doi/10.1103/PhysRevD.93.044057 (visited on 07/27/2025).

[50] Li Yang et al. "Thermoelectric porous laser-induced graphene-based strain-temperature decoupling and self-powered sensing". en. In: *Nature Communications* 16.1 (Jan. 2025). Publisher: Nature Publishing Group, p. 792. ISSN: 2041-1723. DOI: 10.1038/s41467-024-55790-x. URL: https://www.nature.com/articles/s41467-024-55790-x (visited on 05/22/2025).

[51] Yichi Zhang et al. "Magnetic-field tuning of the Casimir force". en. In: *Nature Physics* 20.8 (Aug. 2024). Publisher: Nature Publishing Group, pp. 1282–1287. ISSN: 1745-2481. DOI: 10.1038/s41567-024-02521-0. URL: https://www.nature.com/articles/s41567-024-02521-0 (visited on 05/22/2025).

[52] Ziwen Zhang et al. "Unexpected clustering pattern in dwarf galaxies challenges formation models". en. In: *Nature* (May 2025). Publisher: Nature Publishing Group, pp. 1–6. ISSN: 1476-4687. DOI: 10.1038/s41586-025-08965-5. URL: https://www.nature.com/articles/s41586-025-08965-5 (visited on 05/22/2025).

[53] Zixin Zhang et al. “Computational modelling of the semi-classical quantum vacuum in 3D”. en. In: *Communications Physics* 8.1 (June 2025). Publisher: Nature Publishing Group, pp. 1–11. ISSN: 2399-3650. DOI: 10.1038/s42005-025-02128-8. URL: https://www.nature.com/articles/s42005-025-02128-8 (visited on 06/12/2025).

[54] W. H. Zurek. “Decoherence, einselection, and the quantum origins of the classical”. In: *Reviews of Modern Physics* 75 (2003), pp. 715–775. DOI: 10.1103/RevModPhys.75.715.