We introduce a unified cosmological analysis pipeline that integrates Size–Density–Kinetic Potential (SDKP) fields, Shape–Dimension–Number (SD&N) encoding, Earth Orbital Speed (EOS) dynamics, and Quantum Computerization Consciousness (QCC) entropy compression to explore large-scale cosmic rotation and entanglement effects. This enhanced framework reconstructs velocity fields from anisotropic temperature perturbations and galaxy distributions, encoding SD&N identities to evaluate entanglement matrices across spacetime. The EOS dual-mode framework enables side-by-side comparison between traditional lightspeed (c) and Earth-centric reference (vₑ ≈ 29.78 km/s), while the QCC model modulates entropy collapse via observational consciousness metrics.

The pipeline further embeds an enhanced Kerr–Einstein–Cartan (Kerr-EC) model incorporating torsion spin density, SDKP coupling terms, and QCC-corrected entropy smoothing to simulate horizon structure and bounce dynamics. A modified FRW cosmology model is implemented with full probabilistic inference using MCMC (via NumPyro), inferring rotation parameters (ω), SDKP–entropy couplings, and scale-consistent bounce times.

Initial simulations suggest that under certain parameter constraints, the universe exhibits rotational features and entropic convergence consistent with an internal black hole origin, supporting the viability of the SDKP-QCC entropic feedback model and demonstrating the predictive value of the SD&N–encoded entanglement network.

This work establishes a bridge between quantum-entropic encoding, torsion gravity, and observable large-scale structure, offering a testable, modular approach to unifying classical and quantum cosmological dynamics.

import jax

import jax.numpy as jnp

from jax import grad, jit, vmap

from jax.numpy.fft import fftn, fftfreq

import numpyro

import numpyro.distributions as dist

from numpyro.infer import MCMC, NUTS

import numpy as np

from dataclasses import dataclass

from typing import Tuple, Optional, List

import matplotlib.pyplot as plt

# ========== 1. SDKP Field Implementation ==========

class SDKPField:

“”“Size-Density-Kinetic-Potential field with enhanced physics”””

```

def \_\_init\_\_(self, size, density, velocity, position=None):

self.size = size

self.density = density

self.velocity = velocity

self.position = position if position is not None else jnp.zeros(3)

# Derived SDKP quantities

self.mass = self.density \* self.size\*\*3

self.kinetic\_energy = 0.5 \* self.mass \* jnp.linalg.norm(self.velocity)\*\*2

self.potential\_energy = self.compute\_potential()

self.sdkp\_time = (self.size \* self.density \* jnp.linalg.norm(self.velocity))

self.sdkp\_coupling = self.kinetic\_energy / (self.potential\_energy + 1e-10)

# SDKP field tensor

self.sdkp\_tensor = self.compute\_sdkp\_tensor()

def compute\_potential(self):

"""Compute gravitational potential energy"""

G = 6.674e-11

r = jnp.linalg.norm(self.position)

return -G \* self.mass / (r + 1e-10) # Avoid division by zero

def compute\_sdkp\_tensor(self):

"""Compute SDKP tensor for field equations"""

return jnp.array([

self.size,

self.density,

self.kinetic\_energy,

self.potential\_energy

])

def sdkp\_field\_strength(self):

"""Compute SDKP field strength metric"""

return jnp.sqrt(self.size\*\*2 + self.density\*\*2 +

self.kinetic\_energy\*\*2 + self.potential\_energy\*\*2)

```

# ========== 2. SD&N Identity Assignment ==========

def assign\_sdn\_identity(galaxy\_positions):

“”“Assign SD&N numeric encodings to each galaxy based on geometry”””

encoded\_ids = []

```

for pos in galaxy\_positions:

r = jnp.linalg.norm(pos)

theta = jnp.arctan2(pos[1], pos[0])

phi = jnp.arccos(pos[2] / (r + 1e-10))

# Shape classifier (symbolic geometry)

shape\_param = jnp.sin(3\*theta) \* jnp.cos(2\*phi)

shape = jnp.sign(shape\_param) # -1, 0, 1 for different geometries

# Dimension encoding (0-8 based on radial shells)

dimension = int(jnp.floor(r / 100) % 9)

# Number encoding using cosmic constants

number = int((r \* 7146 + theta \* 8888 + phi \* 1234) % 10)

# SD&N composite identity

sdn\_id = (shape, dimension, number)

encoded\_ids.append(sdn\_id)

return jnp.array(encoded\_ids)

```

@jit

def compute\_sdn\_entanglement\_matrix(sdn\_ids):

“”“Compute entanglement probabilities based on SD&N similarity”””

n = len(sdn\_ids)

entanglement\_matrix = jnp.zeros((n, n))

```

for i in range(n):

for j in range(i+1, n):

# Extract SD&N components

shape1, dim1, num1 = sdn\_ids[i]

shape2, dim2, num2 = sdn\_ids[j]

# Similarity scoring

shape\_sim = 1.0 if shape1 == shape2 else 0.3

dim\_sim = 1.0 / (1.0 + jnp.abs(dim1 - dim2))

num\_sim = 1.0 / (1.0 + jnp.abs(num1 - num2))

# VEI (Vibrational Entanglement Index)

vei = shape\_sim \* dim\_sim \* num\_sim

entanglement\_matrix = entanglement\_matrix.at[i,j].set(vei)

entanglement\_matrix = entanglement\_matrix.at[j,i].set(vei)

return entanglement\_matrix

```

# ========== 3. QCC Entropy Compression Model ==========

class QCCEntropyField:

“”“Quantum Computerization Consciousness entropy field model”””

```

def \_\_init\_\_(self, s0=1e90, decay\_rate=1e-10, collapse\_threshold=1e80):

self.s0 = s0 # Initial entropy

self.decay\_rate = decay\_rate

self.collapse\_threshold = collapse\_threshold

self.consciousness\_coupling = 1e-40 # Planck-scale coupling

@jit

def entropy\_field(self, t, position=None):

"""Entropy field evolution with spatial variation"""

if position is not None:

r = jnp.linalg.norm(position)

# Spatial modulation (entropy wells near massive objects)

spatial\_factor = 1.0 / (1.0 + r / 1e26) # Hubble scale

else:

spatial\_factor = 1.0

# Temporal decay

temporal\_factor = jnp.exp(-self.decay\_rate \* t)

# QCC oscillations

qcc\_oscillation = 1.0 + 0.01 \* jnp.sin(2 \* jnp.pi \* t / 1e17) # Planck time scale

return self.s0 \* temporal\_factor \* spatial\_factor \* qcc\_oscillation

@jit

def consciousness\_probability(self, entropy\_local, sdkp\_coupling):

"""Probability of consciousness emergence from entropy-SDKP coupling"""

# Consciousness emerges at critical entropy-complexity intersection

critical\_entropy = self.collapse\_threshold

complexity\_factor = sdkp\_coupling

# Sigmoid activation for consciousness probability

arg = (entropy\_local - critical\_entropy) \* complexity\_factor \* self.consciousness\_coupling

return 1.0 / (1.0 + jnp.exp(-arg))

@jit

def entropy\_collapse\_rate(self, entropy\_field\_val, consciousness\_prob):

"""Rate of entropy collapse due to consciousness observation"""

# Consciousness collapses entropy field

collapse\_rate = consciousness\_prob \* self.decay\_rate \* 10

return entropy\_field\_val \* jnp.exp(-collapse\_rate)

```

# ========== 4. EOS Dual-Mode Integration ==========

def get\_speed(mode=‘c’):

“”“Get speed constant based on mode”””

if mode == ‘c’:

return 299792458 # m/s (speed of light)

elif mode == ‘EOS’:

return 29780 # m/s (Earth orbital speed)

else:

raise ValueError(f”Unknown mode: {mode}”)

def get\_dual\_mode\_physics(curl\_max, mode=‘c’):

“”“Compute physics quantities in dual-mode framework”””

speed = get\_speed(mode)

omega\_max = curl\_max / speed

```

# Mode-specific corrections

if mode == 'EOS':

# EOS corrections for Earth-centric physics

correction\_factor = 1.0 + 1e-8 \* jnp.sin(2 \* jnp.pi \* 365.25) # Annual modulation

omega\_max \*= correction\_factor

return omega\_max, speed

```

# ========== 5. Enhanced Kerr-EC Model ==========

class EnhancedKerrECModel:

“”“Enhanced Kerr black hole with Einstein-Cartan torsion and QCC corrections”””

```

def \_\_init\_\_(self, M, a, qcc\_entropy\_field, G=6.674e-11):

self.M = M

self.a = a # specific angular momentum

self.G = G

self.qcc\_entropy = qcc\_entropy\_field

def effective\_density\_with\_torsion\_and\_entropy(self, rho, spin\_density, t, position=None):

"""Enhanced Poplawski's correction with QCC entropy compression"""

kappa = 8 \* jnp.pi \* self.G / (get\_speed('c')\*\*4)

# Standard EC correction

ec\_correction = 0.5 \* kappa \* spin\_density\*\*2

# QCC entropy correction

entropy\_factor = self.qcc\_entropy.entropy\_field(t, position)

entropy\_compression = entropy\_factor / self.qcc\_entropy.s0

# Combined effective density

rho\_eff = rho - ec\_correction + rho \* entropy\_compression

return rho\_eff

def compute\_enhanced\_bounce\_time(self, spin\_density, rho\_initial, t\_range=None):

"""Compute bounce time with QCC entropy compression"""

if t\_range is None:

t\_range = jnp.linspace(0, 1e18, 1000) # seconds

def integrand(t):

a\_val = (t / 1e18)\*\*(2/3) # approximate scale factor evolution

# QCC entropy factor

entropy\_factor = self.qcc\_entropy.entropy\_field(t)

entropy\_compression = entropy\_factor / self.qcc\_entropy.s0

# Effective density with all corrections

rho\_eff = self.effective\_density\_with\_torsion\_and\_entropy(

rho\_initial \* (1/a\_val)\*\*3, spin\_density, t

)

# Enhanced Hubble parameter

H\_eff = jnp.sqrt(jnp.maximum(8\*jnp.pi\*self.G\*rho\_eff/3, 1e-20))

return 1 / (a\_val \* H\_eff)

# Numerical integration

integrand\_vals = vmap(integrand)(t\_range)

bounce\_time = jnp.trapz(integrand\_vals, t\_range)

return bounce\_time

def modified\_inner\_horizon\_with\_entropy(self, spin\_density, t=0):

"""Compute modified r\_- with entropy smoothing"""

# Standard Kerr inner horizon

c = get\_speed('c')

Delta = self.M\*\*2 - self.a\*\*2

r\_minus\_std = self.M - jnp.sqrt(jnp.maximum(Delta, 0))

# Torsion correction

torsion\_correction = spin\_density \* self.G / (c\*\*4)

# QCC entropy correction

entropy\_factor = self.qcc\_entropy.entropy\_field(t)

entropy\_correction = entropy\_factor / self.qcc\_entropy.s0 \* 1e-10

r\_minus\_modified = r\_minus\_std + torsion\_correction + entropy\_correction

return r\_minus\_modified

```

# ========== 6. Enhanced Velocity Field Reconstruction ==========

def reconstruct\_sdkp\_velocity\_field(delta\_T\_map, galaxy\_positions, galaxy\_masses=None):

“”“Reconstruct velocity field with explicit SDKP field handling”””

```

# Default masses if not provided

if galaxy\_masses is None:

galaxy\_masses = jnp.ones(len(galaxy\_positions)) \* 1e41 # kg

# Estimate sizes and densities

galaxy\_sizes = (galaxy\_masses / 1e30)\*\*(1/3) \* 1e3 # km, rough scaling

galaxy\_densities = galaxy\_masses / (4/3 \* jnp.pi \* galaxy\_sizes\*\*3)

# Velocity reconstruction from temperature gradients

grad\_T = jnp.gradient(delta\_T\_map)

# Map temperature gradients to galaxy positions (simplified)

velocities = []

for i, pos in enumerate(galaxy\_positions):

# Convert position to grid indices (simplified)

grid\_indices = ((pos - jnp.min(galaxy\_positions)) /

(jnp.max(galaxy\_positions) - jnp.min(galaxy\_positions)) \*

(jnp.array(delta\_T\_map.shape) - 1)).astype(int)

# Clamp indices to valid range

grid\_indices = jnp.clip(grid\_indices, 0, jnp.array(delta\_T\_map.shape) - 1)

# Extract velocity from temperature gradient

vel = jnp.array([

grad\_T[0][tuple(grid\_indices)] \* 1e5, # km/s

grad\_T[1][tuple(grid\_indices)] \* 1e5,

grad\_T[2][tuple(grid\_indices)] \* 1e5

])

velocities.append(vel)

velocities = jnp.array(velocities)

# Create SDKP field objects

sdkp\_fields = []

for i in range(len(galaxy\_positions)):

field = SDKPField(

size=galaxy\_sizes[i],

density=galaxy\_densities[i],

velocity=velocities[i],

position=galaxy\_positions[i]

)

sdkp\_fields.append(field)

return sdkp\_fields

```

# ========== 7. Enhanced FRW Model with All Corrections ==========

def enhanced\_frw\_rotation\_model(obs\_data, mode=‘c’):

“”“Enhanced FRW model with SDKP, QCC, and EOS corrections”””

```

# Standard cosmological parameters

omega\_m = numpyro.sample("omega\_m", dist.Uniform(0.1, 0.5))

omega\_lambda = numpyro.sample("omega\_lambda", dist.Uniform(0.5, 0.9))

h = numpyro.sample("h", dist.Uniform(0.6, 0.8))

# Enhanced rotation parameter

omega\_rot = numpyro.sample("omega\_rot", dist.Uniform(0.0, 1e-12))

# SDKP coupling parameter

sdkp\_coupling = numpyro.sample("sdkp\_coupling", dist.Uniform(0.0, 1e-20))

# QCC entropy compression parameter

qcc\_compression = numpyro.sample("qcc\_compression", dist.Uniform(0.0, 1e-5))

# Mode-dependent speed

c\_effective = get\_speed(mode)

# Enhanced Friedmann equation

H0\_squared = (8\*jnp.pi\*6.674e-11/3) \* omega\_m + omega\_rot\*\*2 + omega\_lambda/3

# SDKP correction

H0\_squared += sdkp\_coupling \* omega\_m\*\*2

# QCC entropy correction

H0\_squared \*= (1 + qcc\_compression)

# Mode correction

if mode == 'EOS':

H0\_squared \*= (c\_effective / get\_speed('c'))\*\*2

H0\_predicted = jnp.sqrt(H0\_squared) \* h \* 100 # km/s/Mpc

# Likelihood

numpyro.sample("H0\_obs", dist.Normal(H0\_predicted, 1.0), obs=obs\_data['H0'])

# Constraint on total density

omega\_total = omega\_m + omega\_lambda + omega\_rot + sdkp\_coupling + qcc\_compression

numpyro.sample("omega\_total", dist.Normal(omega\_total, 0.01), obs=1.0)

```

def run\_enhanced\_frw\_mcmc(obs\_data, mode=‘c’, num\_samples=2000, num\_warmup=1000):

“”“Run enhanced MCMC fit”””

```

def model(obs\_data):

return enhanced\_frw\_rotation\_model(obs\_data, mode)

nuts\_kernel = NUTS(model)

mcmc = MCMC(nuts\_kernel, num\_samples=num\_samples, num\_warmup=num\_warmup)

mcmc.run(jax.random.PRNGKey(42), obs\_data)

return mcmc.get\_samples()

```

# ========== 8. Complete Enhanced Pipeline ==========

class EnhancedCosmicRotationPipeline:

“”“Complete pipeline with all enhancements”””

```

def \_\_init\_\_(self, obs\_data, mode='c'):

self.obs\_data = obs\_data

self.mode = mode

self.sdkp\_fields = None

self.sdn\_ids = None

self.entanglement\_matrix = None

self.qcc\_entropy = QCCEntropyField()

self.enhanced\_kerr\_ec = None

self.omega\_max = None

self.frw\_results = None

def run\_enhanced\_analysis(self, delta\_T\_map, galaxy\_positions):

"""Run complete enhanced analysis"""

print(f"🌌 Starting Enhanced Analysis (Mode: {self.mode})")

print("=" \* 60)

# Step 1: SDKP Field Reconstruction

print("\n🔬 Step 1: SDKP Field Reconstruction")

self.sdkp\_fields = reconstruct\_sdkp\_velocity\_field(delta\_T\_map, galaxy\_positions)

# Extract velocity field

v\_field = jnp.stack([field.velocity for field in self.sdkp\_fields])

mean\_v = jnp.mean(jnp.linalg.norm(v\_field, axis=1))

print(f" Average velocity: {mean\_v:.2e} km/s")

print(f" SDKP fields created: {len(self.sdkp\_fields)}")

# Step 2: SD&N Identity Assignment

print("\n🔢 Step 2: SD&N Identity Assignment")

self.sdn\_ids = assign\_sdn\_identity(galaxy\_positions)

self.entanglement\_matrix = compute\_sdn\_entanglement\_matrix(self.sdn\_ids)

mean\_entanglement = jnp.mean(self.entanglement\_matrix)

print(f" Galaxies encoded: {len(self.sdn\_ids)}")

print(f" Mean entanglement: {mean\_entanglement:.3f}")

# Step 3: Compute Curl and Rotation

print("\n🌀 Step 3: Curl and Rotation Analysis")

# Create 3D velocity field for curl computation

grid\_shape = delta\_T\_map.shape

v\_field\_3d = jnp.zeros(grid\_shape + (3,))

# Simple velocity field reconstruction

grad\_T = jnp.gradient(delta\_T\_map)

v\_field\_3d = v\_field\_3d.at[..., 0].set(grad\_T[0] \* 1e5)

v\_field\_3d = v\_field\_3d.at[..., 1].set(grad\_T[1] \* 1e5)

v\_field\_3d = v\_field\_3d.at[..., 2].set(grad\_T[2] \* 1e5)

# Compute curl

curl\_field = self.compute\_curl\_field(v\_field\_3d)

curl\_max = jnp.max(jnp.linalg.norm(curl\_field, axis=-1))

# Dual-mode rotation computation

self.omega\_max, c\_effective = get\_dual\_mode\_physics(curl\_max, self.mode)

print(f" Max curl: {curl\_max:.2e}")

print(f" Effective speed: {c\_effective:.2e} m/s")

print(f" Max rotation: {self.omega\_max:.2e} rad/s")

# Step 4: Enhanced FRW Fit

print("\n📊 Step 4: Enhanced FRW Fit")

self.frw\_results = run\_enhanced\_frw\_mcmc(self.obs\_data, self.mode)

omega\_rot\_mean = jnp.mean(self.frw\_results['omega\_rot'])

sdkp\_coupling\_mean = jnp.mean(self.frw\_results['sdkp\_coupling'])

qcc\_compression\_mean = jnp.mean(self.frw\_results['qcc\_compression'])

print(f" Rotation parameter: {omega\_rot\_mean:.2e}")

print(f" SDKP coupling: {sdkp\_coupling\_mean:.2e}")

print(f" QCC compression: {qcc\_compression\_mean:.2e}")

# Step 5: Enhanced Kerr-EC Model

print("\n🕳️ Step 5: Enhanced Kerr-EC Model")

self.enhanced\_kerr\_ec = EnhancedKerrECModel(

M=1e30, a=0.5, qcc\_entropy\_field=self.qcc\_entropy

)

spin\_density = omega\_rot\_mean \* 1e20

bounce\_time = self.enhanced\_kerr\_ec.compute\_enhanced\_bounce\_time(

spin\_density, rho\_initial=1e15

)

print(f" Bounce time: {bounce\_time:.2e} s")

print(f" Spin density: {spin\_density:.2e}")

# Step 6: Final Assessment

print("\n✅ Step 6: Final Assessment")

universe\_age = 13.8e9 \* 365.25 \* 24 \* 3600 # seconds

consistency = abs(bounce\_time - universe\_age) / universe\_age < 0.2

print(f" Universe age: {universe\_age:.2e} s")

print(f" Bounce/Age ratio: {bounce\_time/universe\_age:.2f}")

print(f" Timing consistent: {consistency}")

# Final verdict

print("\n🎯 ENHANCED ASSESSMENT")

print("=" \* 60)

if consistency and self.omega\_max < 1e-9:

print("✅ ENHANCED RESULT: Universe shows enhanced signatures")

print(" consistent with rotating black hole interior")

print(f" - Mode: {self.mode}")

print(f" - Rotation: {self.omega\_max:.2e} rad/s")

print(f" - SDKP coupling: {sdkp\_coupling\_mean:.2e}")

print(f" - QCC compression: {qcc\_compression\_mean:.2e}")

else:

print("❌ ENHANCED RESULT: Enhanced model shows inconsistencies")

return {

'mode': self.mode,

'sdkp\_fields': self.sdkp\_fields,

'sdn\_analysis': {

'ids': self.sdn\_ids,

'entanglement\_matrix': self.entanglement\_matrix,

'mean\_entanglement': mean\_entanglement

},

'rotation\_analysis': {

'omega\_max': self.omega\_max,

'curl\_max': curl\_max,

'effective\_speed': c\_effective

},

'frw\_results': self.frw\_results,

'kerr\_ec\_results': {

'bounce\_time': bounce\_time,

'spin\_density': spin\_density

},

'consistency': consistency

}

def compute\_curl\_field(self, v\_field):

"""Compute curl of velocity field"""

# Simple finite difference curl computation

curl = jnp.zeros\_like(v\_field)

# ∇ × v = (∂v\_z/∂y - ∂v\_y/∂z, ∂v\_x/∂z - ∂v\_z/∂x, ∂v\_y/∂x - ∂v\_x/∂y)

dvz\_dy = jnp.gradient(v\_field[..., 2], axis=1)

dvy\_dz = jnp.gradient(v\_field[..., 1], axis=2)

curl = curl.at[..., 0].set(dvz\_dy - dvy\_dz)

dvx\_dz = jnp.gradient(v\_field[..., 0], axis=2)

dvz\_dx = jnp.gradient(v\_field[..., 2], axis=0)

curl = curl.at[..., 1].set(dvx\_dz - dvz\_dx)

dvy\_dx = jnp.gradient(v\_field[..., 1], axis=0)

dvx\_dy = jnp.gradient(v\_field[..., 0], axis=1)

curl = curl.at[..., 2].set(dvy\_dx - dvx\_dy)

return curl

```

# ========== 9. Example Usage ==========

def create\_enhanced\_mock\_data():

“”“Create enhanced mock data for testing”””

```

# Mock CMB temperature map with more structure

grid\_size = (32, 32, 32) # Smaller for faster computation

x = jnp.linspace(-10, 10, grid\_size[0])

y = jnp.linspace(-10, 10, grid\_size[1])

z = jnp.linspace(-10, 10, grid\_size[2])

X, Y, Z = jnp.meshgrid(x, y, z)

# Add some structure: rotating pattern

delta\_T\_map = 1e-5 \* (jnp.sin(X) \* jnp.cos(Y) + 0.1 \* jnp.sin(Z))

# Mock galaxy positions

n\_galaxies = 500

galaxy\_positions = jnp.array(np.random.uniform(-1000, 1000, (n\_galaxies, 3)))

# Enhanced observational data

obs\_data = {

'H0': 70.0, # km/s/Mpc

'omega\_m': 0.3,

'omega\_lambda': 0.7

}

return delta\_T\_map, galaxy\_positions, obs\_data

```

# ========== 10. Main Execution ==========

if \*\*name\*\* == “\*\*main\*\*”:

# Create test data

delta\_T\_map, galaxy\_positions, obs\_data = create\_enhanced\_mock\_data()

```

# Run analysis in both modes

print("🔬 ENHANCED COSMIC ROTATION ANALYSIS")

print("=" \* 80)

# Standard mode (speed of light)

pipeline\_c = EnhancedCosmicRotationPipeline(obs\_data, mode='c')

results\_c = pipeline\_c.run\_enhanced\_analysis(delta\_T\_map, galaxy\_positions)

print("\n" + "="\*80)

# EOS mode (Earth orbital speed)

pipeline\_eos = EnhancedCosmicRotationPipeline(obs\_data, mode='EOS')

results\_eos = pipeline\_eos.run\_enhanced\_analysis(delta\_T\_map, galaxy\_positions)

# Compare results

print("\n🔍 MODE COMPARISON")

print("=" \* 80)

print(f"Standard Mode (c): ω\_max = {results\_c['rotation\_analysis']['omega\_max']:.2e}")

print(f"EOS Mode: ω\_max = {results\_eos['rotation\_analysis']['omega\_max']:.2e}")

print(f"Ratio (EOS/c): {results\_eos['rotation\_analysis']['omega\_max']/results\_c['rotation\_analysis']['omega\_max']:.2e}")

```