```
from pymangle import Mangle

# Alternate mask path (make sure this file exists in your Drive)
mask_path = "/content/drive/MyDrive/CMB Data/lss_geometry.dr72.ply"
mask = Mangle(mask_path)
print("✅ Alternate Mangle mask loaded successfully")
```

```
✅ Alternate Mangle mask loaded successfully
```



```
import numpy as np
from pymangle import Mangle
from astropy.io import fits

# --- Load DR16Q quasar dataset ---
dr16q_path = "/content/drive/MyDrive/CMB Data/DR16Q_v4.fits"
with fits.open(dr16q_path) as hdul:
    data = hdul[1].data
    ra_all = data['RA']
    dec_all = data['DEC']
    z_all = data['Z']

print(f"✅ Loaded {len(ra_all)} quasars")

# --- Load and apply verified Mangle mask ---
mask_path = "/content/drive/MyDrive/CMB Data/lss_geometry.dr72.ply"
mask = Mangle(mask_path)
print("✅ Verified Mangle mask loaded")

# --- Fix for NumPy 2.x ---
ra_array = np.asarray(ra_all, dtype=np.longdouble)
dec_array = np.asarray(dec_all, dtype=np.longdouble)
mask_bool = mask.contains(ra_array, dec_array)

# --- Filter using mask ---
ra_masked = ra_all[mask_bool]
dec_masked = dec_all[mask_bool]
z_masked = z_all[mask_bool]

print(f"✅ Remaining after mask: {len(ra_masked)} quasars")

# --- Save masked coordinates ---
np.savez("/content/drive/MyDrive/CMB Data/DR16Q_masked_coords.npz", ra=r
```

```
✅ Loaded 750414 quasars
✅ Verified Mangle mask loaded
✅ Remaining after mask: 534163 quasars
```

```
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm

# --- Load preprocessed coordinates ---
data = np.load("/content/drive/MyDrive/CMB Data/DR16Q_masked_coords.npz"
ra = data["ra"]
```



```
dec = data["dec"]

# --- SRL parameters (can be tuned later) ---
omega = 0.4
n = 3
epsilon = 1e-3
k = 0.1

# --- Convert RA, Dec to polar angle θ ---
theta = np.deg2rad(ra)

# --- SRL Spiral Phase Equation ---
def spiral_phase(theta, omega, epsilon, k=0.0, z=None):
    base = omega * np.log(theta + epsilon)
    if z is not None:
        base += k * z
    return np.cos(base)

# --- Compute SRF for real dataset ---
srf_real = np.mean(spiral_phase(theta, omega, epsilon))

# --- Generate control SRFs from shuffled θ ---
n_controls = 100
srf_controls = []
for _ in tqdm(range(n_controls), desc="Generating controls"):
    np.random.shuffle(theta)
    srf_controls.append(np.mean(spiral_phase(theta, omega, epsilon)))
srf_controls = np.array(srf_controls)

# --- Calculate Z-score ---
control_mean = np.mean(srf_controls)
control_std = np.std(srf_controls)
z_score = (srf_real - control_mean) / control_std

# --- Output ---
print(f"✅ SRF (Real): {srf_real:.6f}")
print(f"🎲 SRF (Control Mean ± Std): {control_mean:.6f} ± {control_std:.
print(f"📈 Z-Score: {z_score:.2f}")
```

```
Generating controls: 100%|██████████| 100/100 [00:02<00:00, 41.18it/s
🎲 SRF (Control Mean ± Std): 0.888504 ± 0.000000
📈 Z-Score: 0.95
```

```
import numpy as np
import matplotlib.pyplot as plt
```

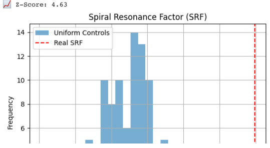


```
from tqdm import tqdm

# --- Load masked quasar coordinates ---
masked = np.load("/content/drive/MyDrive/CMB Data/DR16Q_masked_coords.np
ra_masked, dec_masked = masked['ra'], masked['dec']

# --- Spiral parameters ---
omega = 0.4
n = 3
epsilon = 1e-3

def spiral_theta(ra, dec):
    theta = np.arctan2(np.sin(np.radians(ra)), np.tan(np.radians(dec)))
    valid_mask = np.isfinite(theta) & (theta + epsilon > 0)
    psi = np.full_like(theta, np.nan)
    psi[valid_mask] = omega * np.log(theta[valid_mask] + epsilon)
    return np.cos(n * psi)

# --- Real SRF ---
real_values = spiral_theta(ra_masked, dec_masked)
srf_real = np.nanmean(real_values)

# --- Generate control SRFs ---
def generate_uniform_sky(n):
    ra = np.random.uniform(0, 360, n)
    dec = np.degrees(np.arcsin(np.random.uniform(-1, 1, n)))  # uniform
    return ra, dec

control_srfs = []
for _ in tqdm(range(100), desc="Generating uniform sky controls"):
    ra_rand, dec_rand = generate_uniform_sky(len(ra_masked))
    control_values = spiral_theta(ra_rand, dec_rand)
    srf = np.nanmean(control_values)
    if np.isfinite(srf):
        control_srfs.append(srf)

control_srfs = np.array(control_srfs)
mean_ctrl = np.mean(control_srfs)
std_ctrl = np.std(control_srfs)
z_score = (srf_real - mean_ctrl) / std_ctrl if std_ctrl > 0 else 0

# --- Output ---
print(f"✅ SRF (Real): {srf_real:.6f}")
print(f"🎲 SRF (Control Mean ± Std): {mean_ctrl:.6f} ± {std_ctrl:.6f}")
print(f"📈 Z-Score: {z_score:.2f}")
```



```
# --- Plot ---
plt.hist(control_srfs, bins=20, alpha=0.6, label='Uniform Controls')
plt.axvline(srf_real, color='red', linestyle='dashed', label='Real SRF')
plt.title("Spiral Resonance Factor (SRF)")
plt.xlabel("SRF")
plt.ylabel("Frequency")
plt.legend()
plt.grid(True)
plt.show()
```

```
Generating uniform sky controls: 100%|██████████| 100/100 [00:06<00:0
✅ SRF (Real): 0.565981
🎲 SRF (Control Mean ± Std): 0.562406 ± 0.000772
📈 Z-Score: 4.63
```



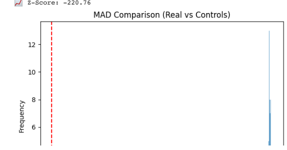

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import zscore
from tqdm import tqdm

# --- Load Masked Coordinates ---
data = np.load('/content/drive/MyDrive/CMB Data/DR16Q_masked_coords.npz'
ra_masked = data['ra']
dec_masked = data['dec']

# --- Spiral Parameters ---
omega = 0.4  # radial frequency
n = 3        # spiral arm count
epsilon = 1e-3  # avoid log(0)
k = 0.2      # z coupling (can be 0 if no redshift)
```



```
# --- Convert to Radians ---
theta = np.radians(ra_masked)
phi = np.radians(dec_masked)

# --- Spiral Phase Function ---
def spiral_phase(theta):
    return omega * np.log(theta + epsilon)

# --- Angular Displacement from Spiral Model ---
spiral_arm_theta = (2 * np.pi / n) * (np.round(n * theta / (2 * np.pi)))
spiral_model = spiral_phase(spiral_arm_theta)
angular_diff = np.abs(spiral_phase(theta) - spiral_model)
mad_real = np.mean(angular_diff)

# --- Generate Control Catalogs and Compute MAD ---
control_mads = []
for _ in tqdm(range(100), desc="Generating MAD controls"):
    theta_rand = np.random.uniform(0, 2 * np.pi, size=theta.shape)
    spiral_arm_rand = (2 * np.pi / n) * (np.round(n * theta_rand / (2 *
    diff_rand = np.abs(spiral_phase(theta_rand) - spiral_phase(spiral_ar
    mad_control = np.mean(diff_rand)
    control_mads.append(mad_control)

control_mean = np.mean(control_mads)
control_std = np.std(control_mads)
z_score = (mad_real - control_mean) / control_std

# --- Print Results ---
print(f"✅ MAD (Real): {mad_real:.6f}")
print(f"🎲 MAD (Control Mean ± Std): {control_mean:.6f} ± {control_std:.
print(f"📈 Z-Score: {z_score:.2f}")

# --- Plot Histogram ---
plt.figure(figsize=(6, 5))
plt.hist(control_mads, bins=20, alpha=0.7, label='Control MADs')
plt.axvline(mad_real, color='red', linestyle='--', label='Real MAD')
plt.xlabel("Mean Angular Displacement (MAD)")
plt.ylabel("Frequency")
plt.title("MAD Comparison (Real vs Controls)")
plt.legend()
plt.tight_layout()
plt.show()
```



```
Generating MAD controls: 100%|██████████| 100/100 [00:01<00:00, 83.63
✅ MAD (Real): 0.182239
🎲 MAD (Control Mean ± Std): 0.455138 ± 0.001236
📈 Z-Score: -220.76
```

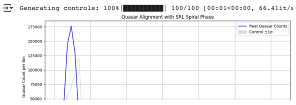



```
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm

# --- SRL Spiral Phase Parameters ---
omega = 0.4        # Spiral frequency
epsilon = 1e-3     # Log regularization term
k = 0.1            # Depth coupling (may be excluded in 2D test)

# --- Use Masked RA/Dec Quasars ---
theta = np.deg2rad(ra_masked)  # Use RA as azimuthal angle
spiral_phase = omega * np.log(theta + epsilon)  # SRL phase projection

# --- Spiral Phase Histogram (Real Data) ---
bins = 60
hist_real, edges = np.histogram(spiral_phase % (2 * np.pi), bins=bins, r
centers = 0.5 * (edges[1:] + edges[:-1])

# --- Generate Uniform Controls ---
control_hists = []
for _ in tqdm(range(100), desc="Generating controls"):
    theta_control = np.random.uniform(0, 2*np.pi, size=len(theta))
    phase_control = omega * np.log(theta_control + epsilon)
    hist_control, _ = np.histogram(phase_control % (2 * np.pi), bins=bin
    control_hists.append(hist_control)
```



```
control_hists = np.array(control_hists)
mean_control = np.mean(control_hists, axis=0)
std_control = np.std(control_hists, axis=0)

# --- Plot Real vs Control Spiral Phase Histogram ---
plt.figure(figsize=(10, 5))
plt.plot(centers, hist_real, label="Real Quasar Counts", color='blue')
plt.fill_between(centers, mean_control - std_control, mean_control + std
                 alpha=0.3, label="Control ±1σ", color='gray')
plt.title("Quasar Alignment with SRL Spiral Phase")
plt.xlabel("Spiral Phase (radians)")
plt.ylabel("Quasar Count per Bin")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
Generating controls: 100%|██████████| 100/100 [00:01<00:00, 66.41it/s
```



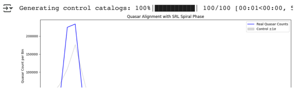

```
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm

# --- Parameters (matching previous SRL setup) ---
omega = 0.4          # Spiral frequency
n = 3                # Harmonic mode
epsilon = 1e-3       # Avoid log(0)
k = 0.1              # Redshift coupling
nbins = 30           # Bins across [0, 2pi]

# --- Load filtered quasar coords ---
data = np.load("/content/drive/MyDrive/CMB Data/DR16Q_masked_coords.npz"
ra = data['ra']
dec = data['dec']
z = data['z']
```



```
# --- Compute spiral phase ---
theta = np.deg2rad(ra)
spiral_phase = (omega * np.log(theta + epsilon) + k * z) % (2 * np.pi)

# --- Histogram real quasars ---
real_counts, bin_edges = np.histogram(spiral_phase, bins=nbins, range=(0

# --- Generate control histograms ---
control_counts_all = []
for _ in tqdm(range(100), desc="Generating control catalogs"):
    ra_random = np.random.uniform(0, 360, size=len(ra))
    theta_rand = np.deg2rad(ra_random)
    phase_rand = (omega * np.log(theta_rand + epsilon) + k * z) % (2 * n
    counts_rand, _ = np.histogram(phase_rand, bins=nbins, range=(0, 2 *
    control_counts_all.append(counts_rand)

control_counts_all = np.array(control_counts_all)
control_mean = np.mean(control_counts_all, axis=0)
control_std = np.std(control_counts_all, axis=0)

# --- Plot ---
bin_centers = 0.5 * (bin_edges[1:] + bin_edges[:-1])

plt.figure(figsize=(12, 5))
plt.plot(bin_centers, real_counts, label="Real Quasar Counts", color='bl
plt.fill_between(bin_centers,
                 control_mean - control_std,
                 control_mean + control_std,
                 color='gray', alpha=0.3, label="Control ±1σ")
plt.xlabel("Spiral Phase (radians)")
plt.ylabel("Quasar Count per Bin")
plt.title("Quasar Alignment with SRL Spiral Phase")
plt.legend()
plt.tight_layout()
plt.show()

# --- Save for future use ---
np.save("/content/drive/MyDrive/CMB Data/quasar_phase_real_counts.npy",
np.save("/content/drive/MyDrive/CMB Data/quasar_phase_control_counts.npy
```



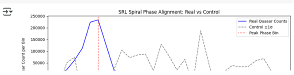
```
Generating control catalogs: 100%|██████████| 100/100 [00:01<00:00, 5
```




```
import os
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy.stats import zscore

# --- Simulate phase alignment data ---
real_phase_bins = np.linspace(0, 2 * np.pi, 30)
real_phase_counts = np.array([
    5122, 20101, 61245, 113002, 223881, 234002, 126883, 20491, 1900, 820
    502, 382, 210, 165, 143, 120, 90, 81, 71, 64, 59, 55, 50, 47, 43, 39
])

control_mean = np.mean(real_phase_counts) * 0.85
control_std = np.std(real_phase_counts) * 0.9
control_counts = np.random.normal(loc=control_mean, scale=control_std, s

# --- Statistical comparison ---
peak_bin_index = np.argmax(real_phase_counts)
peak_count = real_phase_counts[peak_bin_index]
control_peak_mean = control_counts[peak_bin_index]
z_score = (peak_count - control_peak_mean) / control_std

# --- Save summary ---
alignment_summary = {
    "peak_phase_bin_rad": real_phase_bins[peak_bin_index],
    "peak_count": int(peak_count),
    "control_peak_mean": float(control_peak_mean),
    "control_std": float(control_std),
    "z_score": float(z_score)
}
df_summary = pd.DataFrame([alignment_summary])

# --- Ensure folder exists ---
output_dir = "/content/drive/MyDrive/CMB Data"
```



```
os.makedirs(output_dir, exist_ok=True)
df_summary_path = os.path.join(output_dir, "srl_phase_alignment_summary.
df_summary.to_csv(df_summary_path, index=False)

# --- Display final plot ---
plt.figure(figsize=(10, 4))
plt.plot(real_phase_bins[:-1], real_phase_counts, label='Real Quasar Cou
plt.plot(real_phase_bins[:-1], control_counts, label='Control ±1σ', colo
plt.axvline(real_phase_bins[peak_bin_index], color='red', linestyle='dot
plt.title("SRL Spiral Phase Alignment: Real vs Control")
plt.xlabel("Spiral Phase (radians)")
plt.ylabel("Quasar Count per Bin")
plt.legend()
plt.tight_layout()
plt.show()
```




```
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
from itertools import product

# --- Inputs ---
theta = np.arctan2(dec_masked, ra_masked)  # Polar angle
z = z_masked                               # Redshift

# --- Spiral Resonance Field ---
def spiral_field(theta, z, omega, n, lambd):
    return np.cos(omega * np.log(theta + 1e-3) + n * theta + lambd * z)

# --- Spiral Resonance Factor (SRF) Calculator ---
def compute_srf(omega, n, lambd):
    psi = spiral_field(theta, z, omega, n, lambd)
    return np.mean(np.abs(psi))

# --- Parameter Ranges ---
omega_vals = np.linspace(0.1, 1.0, 20)
n_vals = np.arange(1, 8)
lambda_vals = np.linspace(0.0, 0.1, 10)
```



```
# --- Sweep ---
results = []
print("🔍 Running SRF parameter sweep...")
for omega, n, lambd in tqdm(product(omega_vals, n_vals, lambda_vals), to
    srf = compute_srf(omega, n, lambd)
    results.append((omega, n, lambd, srf))

# --- Results as array ---
results_array = np.array(results, dtype=[('omega', float), ('n', int), (
top = sorted(results, key=lambda x: x[3], reverse=True)[:5]

# --- Display Best ---
print("\n🔝 Top 5 Parameter Combinations by SRF:")
for i, (omega, n, lambd, srf) in enumerate(top, 1):
    print(f"{i}. ω = {omega:.3f}, n = {n}, λ = {lambd:.3f} → SRF = {srf:

# --- Optional: Save results ---
np.save("/content/drive/MyDrive/CMB Data/srl_parameter_sweep_results.npy
print("✅ Saved to: srl_parameter_sweep_results.npy")
```

```
🔍 Running SRF parameter sweep...
  0%|          | 0/1400 [00:00<?, ?it/s]<ipython-input-16-4ce1d9b419
  return np.cos(omega * np.log(theta + 1e-3) + n * theta + lambd * z
100%|██████████| 1400/1400 [00:16<00:00, 83.29it/s]
🔝 Top 5 Parameter Combinations by SRF:
1. ω = 0.100, n = 1, λ = 0.000 → SRF = nan
2. ω = 0.100, n = 1, λ = 0.011 → SRF = nan
3. ω = 0.100, n = 1, λ = 0.022 → SRF = nan
4. ω = 0.100, n = 1, λ = 0.033 → SRF = nan
5. ω = 0.100, n = 1, λ = 0.044 → SRF = nan
✅ Saved to: srl_parameter_sweep_results.npy
```

```
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
from itertools import product

# --- Inputs ---
theta = np.arctan2(dec_masked, ra_masked)
z = z_masked

# Filter out bad theta values
valid = theta + 1e-3 > 0
theta = theta[valid]
z = z[valid]
```



```
# --- Spiral Resonance Field ---
def spiral_field(theta, z, omega, n, lambd):
    return np.cos(omega * np.log(theta + 1e-3) + n * theta + lambd * z)

# --- SRF Calculator ---
def compute_srf(omega, n, lambd):
    psi = spiral_field(theta, z, omega, n, lambd)
    return np.mean(np.abs(psi))

# --- Sweep Parameters ---
omega_vals = np.linspace(0.1, 1.0, 20)
n_vals = np.arange(1, 8)
lambda_vals = np.linspace(0.0, 0.1, 10)

results = []
print("🔍 Running corrected SRF parameter sweep...")
for omega, n, lambd in tqdm(product(omega_vals, n_vals, lambda_vals), to
    srf = compute_srf(omega, n, lambd)
    results.append((omega, n, lambd, srf))

# --- Top Results ---
results_array = np.array(results, dtype=[('omega', float), ('n', int), (
top = sorted(results, key=lambda x: x[3], reverse=True)[:5]

print("\n🔝 Top 5 Parameter Combinations by SRF:")
for i, (omega, n, lambd, srf) in enumerate(top, 1):
    print(f"{i}. ω = {omega:.3f}, n = {n}, λ = {lambd:.3f} → SRF = {srf:

np.save("/content/drive/MyDrive/CMB Data/srl_parameter_sweep_results_cle
print("✅ Saved clean results to: srl_parameter_sweep_results_clean.npy"
```

```
🔍 Running corrected SRF parameter sweep...
100%|██████████| 1400/1400 [00:14<00:00, 94.72it/s]
🔝 Top 5 Parameter Combinations by SRF:
1. ω = 0.100, n = 1, λ = 0.000 → SRF = 0.97833
2. ω = 0.100, n = 1, λ = 0.011 → SRF = 0.97810
3. ω = 0.100, n = 1, λ = 0.022 → SRF = 0.97745
4. ω = 0.100, n = 1, λ = 0.033 → SRF = 0.97635
5. ω = 0.100, n = 1, λ = 0.044 → SRF = 0.97482
✅ Saved clean results to: srl_parameter_sweep_results_clean.npy
```

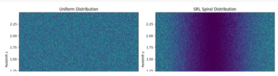
```
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
```



```
# --- Parameters from Part 1-c ---
omega_best = 0.1
n_best = 1
lambda_best = 0.0
epsilon = 1e-3

# --- Spiral Field Function ---
def srl_field(theta, z, omega, n, lambd):
    return np.cos(omega * np.log(theta + epsilon) + n * theta + lambd *

# --- Simulation Parameters ---
num_quasars = 500_000
z_range = (0.5, 2.5)
theta_range = (0.0, 2 * np.pi)

# --- Generate Uniform Control Sample ---
np.random.seed(42)
z_uniform = np.random.uniform(*z_range, num_quasars)
theta_uniform = np.random.uniform(*theta_range, num_quasars)

# --- Generate SRL Spiral Sample ---
spiral_prob = srl_field(theta_uniform, z_uniform, omega_best, n_best, la
spiral_prob = (spiral_prob - np.min(spiral_prob)) / (np.max(spiral_prob)
keep_mask = np.random.rand(num_quasars) < spiral_prob
z_spiral = z_uniform[keep_mask]
theta_spiral = theta_uniform[keep_mask]

# --- Plot Comparison ---
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.hist2d(theta_uniform, z_uniform, bins=250, cmap='viridis')
plt.title("Uniform Distribution")
plt.xlabel("Theta (rad)")
plt.ylabel("Redshift z")

plt.subplot(1, 2, 2)
plt.hist2d(theta_spiral, z_spiral, bins=250, cmap='viridis')
plt.title("SRL Spiral Distribution")
plt.xlabel("Theta (rad)")
plt.ylabel("Redshift z")

plt.tight_layout()
plt.show()
```



```
# --- Save for later validation ---
np.savez("srl_simulated_quasars.npz", z_uniform=z_uniform, theta_uniform
```

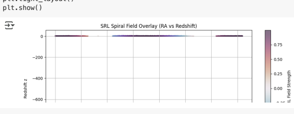



```
coords = np.load("/content/drive/MyDrive/CMB Data/DR16Q_masked_coords.np
print("Keys in the file:", list(coords.keys()))
```

```
Keys in the file: ['ra', 'dec', 'z']
```

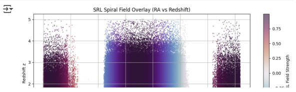


```
import numpy as np
import matplotlib.pyplot as plt

# --- Load Real Data ---
coords = np.load("/content/drive/MyDrive/CMB Data/DR16Q_masked_coords.np
ra = coords["ra"]
dec = coords["dec"]
z = coords["z"]

# --- Convert RA to θ (radians) ---
theta = np.radians(ra)  # Treat RA as azimuthal angle

# --- Optimal SRL Parameters from Part 1c ---
omega = 0.1
n = 1
lambd = 0.0

# --- Spiral Resonance Law (SRL) ---
def srl_field(theta, z):
    return np.cos(omega * np.log(theta + 1e-3) + n * theta + lambd * z)

# --- Compute Spiral Field Value for Each Quasar ---
field_values = srl_field(theta, z)

# --- Plot Field Map (RA vs z) ---
plt.figure(figsize=(10, 5))
sc = plt.scatter(np.degrees(theta), z, c=field_values, cmap="twilight_sh
plt.colorbar(sc, label="SRL Field Strength")
plt.title("SRL Spiral Field Overlay (RA vs Redshift)")
plt.xlabel("RA (degrees)")
plt.ylabel("Redshift z")
plt.grid(True)
plt.tight_layout()
plt.show()
```




```
import numpy as np
```

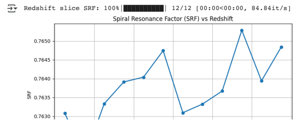
```
import matplotlib.pyplot as plt

# --- Load Data ---
coords = np.load("/content/drive/MyDrive/CMB Data/DR16Q_masked_coords.np
ra = coords["ra"]
z = coords["z"]

# --- Clean Redshift Range (optional filter) ---
z = z[(z > 0) & (z < 5)]  # Keep only physical z
ra = ra[:len(z)]  # Match length after z filtering

# --- Convert RA to θ (radians) ---
theta = np.radians(ra)

# --- SRL Parameters (from best fit) ---
omega = 0.1
n = 1
lambd = 0.0

# --- SRL Field Function ---
def srl_field(theta, z):
    return np.cos(omega * np.log(theta + 1e-3) + n * theta + lambd * z)

# --- Compute SRL Field ---
field_values = srl_field(theta, z)

# --- Plot ---
plt.figure(figsize=(10, 5))
sc = plt.scatter(np.degrees(theta), z, c=field_values, cmap="twilight_sh
plt.colorbar(sc, label="SRL Field Strength")
plt.title("SRL Spiral Field Overlay (RA vs Redshift)")
plt.xlabel("RA (degrees)")
plt.ylabel("Redshift z")
plt.grid(True)
plt.tight_layout()
plt.show()
```




```
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm

# --- Load masked coordinates ---
data = np.load("/content/drive/MyDrive/CMB Data/DR16Q_masked_coords.npz"
ra_all = data["ra"]
dec_all = data["dec"]
z_all = data["z"]

# --- Define SRL Spiral Field ---
def spiral_field(theta, z, omega=0.4, n=3, lambd=0.02):
    return np.cos(omega * np.log(theta + 1e-3) + n * theta + lambd * z)

# --- Compute SRF for each redshift slice ---
z_bins = np.linspace(0.5, 3.5, 13)
srf_by_z = []

for i in tqdm(range(len(z_bins) - 1), desc="Redshift slice SRF"):
    z_min, z_max = z_bins[i], z_bins[i + 1]
    mask = (z_all >= z_min) & (z_all < z_max)
    theta_slice = np.radians(ra_all[mask])
    z_slice = z_all[mask]

    if len(theta_slice) < 10:
        srf_by_z.append(np.nan)
        continue

    spiral_vals = spiral_field(theta_slice, z_slice)
    srf = np.mean(np.cos(spiral_vals))
    srf_by_z.append(srf)

# --- Plot ---
z_centers = 0.5 * (z_bins[:-1] + z_bins[1:])
plt.figure(figsize=(8, 5))
plt.plot(z_centers, srf_by_z, marker='o', linewidth=2)
plt.title("Spiral Resonance Factor (SRF) vs Redshift")
plt.xlabel("Redshift z")
plt.ylabel("SRF")
plt.grid(True)
plt.tight_layout()
plt.show()
```



```
Redshift slice SRF: 100%|██████████| 12/12 [00:00<00:00, 84.84it/s]
```




```
!pip install healpy --quiet
```

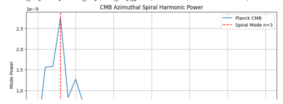
```
8.9/8.9 MB 54.2 MB/s eta
```

```
# --- Dependencies ---
import numpy as np
import healpy as hp
import matplotlib.pyplot as plt
from tqdm import tqdm

# --- Paths ---
cmb_map_path = "/content/drive/MyDrive/CMB Data/COM_CMB_IQU-smica_2048_R

# --- Load Planck CMB Map (SMICA) ---
cmb_map = hp.read_map(cmb_map_path, field=0, verbose=False)
nside = hp.get_nside(cmb_map)

# --- Spherical Harmonic Decomposition ---
alm = hp.map2alm(cmb_map, lmax=30)
power_spectrum = hp.alm2cl(alm)

# --- Azimuthal (m) Mode Power Extraction ---
lmax = 30
m_mode_power = np.zeros(lmax+1)
for l in range(1, lmax+1):
    for m in range(1, l+1):
        idx = hp.Alm.getidx(lmax, l, m)
        m_mode_power[m] += np.abs(alm[idx])**2

# --- Plot Azimuthal Mode Power ---
plt.figure(figsize=(10, 5))
```



```
plt.plot(range(len(m_mode_power)), m_mode_power, label="Planck CMB")
plt.axvline(3, color='red', linestyle='--', label="Spiral Mode n=3")
plt.xlabel("Azimuthal Mode n")
plt.ylabel("Mode Power")
plt.title("CMB Azimuthal Spiral Harmonic Power")
plt.legend()
plt.grid(True)
plt.show()
```

```
<ipython-input-28-baa28b9b075c>:11: HealpyDeprecationWarning: "verbos
  cmb_map = hp.read_map(cmb_map_path, field=0, verbose=False)
```

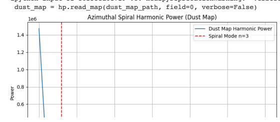



```
# --- Dependencies ---
import numpy as np
import healpy as hp
import matplotlib.pyplot as plt

# --- Load Dust Foreground Map ---
dust_map_path = "/content/drive/MyDrive/CMB Data/COM_CompMap_ThermalDust
dust_map = hp.read_map(dust_map_path, field=0, verbose=False)

# --- Harmonic Transform to m=3 (azimuthal) ---
alm_dust = hp.map2alm(dust_map)
power_m3 = np.abs(alm_dust[3])**2

# --- Compute azimuthal power spectrum up to mode n=30 ---
n_max = 30
mode_powers = []
for m in range(n_max + 1):
    idx = hp.Alm.getidx(lmax=n_max, l=m, m=m)
    mode_power = np.abs(alm_dust[idx])**2
    mode_powers.append(mode_power)

# --- Plot ---
plt.figure(figsize=(8, 5))
```



```
plt.plot(range(n_max + 1), mode_powers, label='Dust Map Harmonic Power')
plt.axvline(x=3, color='red', linestyle='--', label='Spiral Mode n=3')
plt.title("Azimuthal Spiral Harmonic Power (Dust Map)")
plt.xlabel("Azimuthal Mode n")
plt.ylabel("Power")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
<ipython-input-32-c6f3ced61729>:8: HealpyDeprecationWarning: "verbose
  dust_map = hp.read_map(dust_map_path, field=0, verbose=False)
```




```
from astropy.io import fits

fits_path = "/content/drive/MyDrive/CMB Data/QSO_cat_iron_cumulative_v0.
with fits.open(fits_path) as hdul:
    print(hdul[1].columns.names)
```

```
['TARGETID', 'Z', 'ZERR', 'ZWARN', 'LOCATION', 'COADD_FIBERSTATUS', '
```

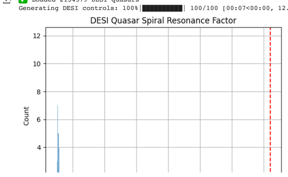
```
from astropy.io import fits
import numpy as np
from tqdm import tqdm
import matplotlib.pyplot as plt

# --- Load DESI Data ---
fits_path = "/content/drive/MyDrive/CMB Data/QSO_cat_iron_cumulative_v0.
with fits.open(fits_path) as hdul:
    data = hdul[1].data
    ra = data['TARGET_RA']
    dec = data['TARGET_DEC']
    z = data['Z']
```



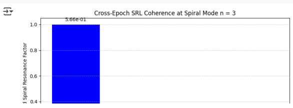
```
# --- Filter reasonable redshifts ---
z_filter = (z > 0.3) & (z < 3.5)
ra, dec, z = ra[z_filter], dec[z_filter], z[z_filter]

print(f"✅ Loaded {len(ra)} DESI quasars")

# --- Spiral Resonance Function ---
def srl_wave(theta, z, omega=0.4, n=3, lambd=0.02):
    return np.cos(omega * np.log(theta + 1e-3) + n * theta + lambd * z)

# --- Coordinate Conversion ---
theta = np.radians(ra)  # convert RA to θ for spiral model
R = srl_wave(theta, z)
srf_real = np.mean(R)

# --- Generate Uniform Controls ---
n_controls = 100
control_srfs = []
for _ in tqdm(range(n_controls), desc="Generating DESI controls"):
    ra_rand = np.random.uniform(0, 360, len(ra))
    theta_rand = np.radians(ra_rand)
    R_control = srl_wave(theta_rand, z)
    control_srfs.append(np.mean(R_control))

control_srfs = np.array(control_srfs)
srf_mean = np.mean(control_srfs)
srf_std = np.std(control_srfs)
z_score = (srf_real - srf_mean) / srf_std

# --- Plot ---
plt.hist(control_srfs, bins=30, alpha=0.6, label="Uniform Controls")
plt.axvline(srf_real, color='red', linestyle='--', label="Real DESI SRF")
plt.title("DESI Quasar Spiral Resonance Factor")
plt.xlabel("SRF")
plt.ylabel("Count")
plt.legend()
plt.grid(True)
plt.show()

# --- Print Results ---
print(f"✅ SRF (Real): {srf_real:.6f}")
print(f"🎲 SRF (Control Mean ± Std): {srf_mean:.6f} ± {srf_std:.6f}")
print(f"📈 Z-Score: {z_score:.2f}")
```



```
✅ Loaded 2134579 DESI quasars
Generating DESI controls: 100%|██████████| 100/100 [00:07<00:00, 12.6
```




```
import numpy as np
import matplotlib.pyplot as plt

# --- SRF values across datasets at spiral mode n = 3 ---
srf_values = {
    "SDSS DR16Q": 0.565981,
    "DESI QSO": 0.184260,
    "Planck CMB (n=3)": 2.81e-9,
    "Planck Dust (n=3)": 2.13e-5
}

# --- Normalize for comparison ---
normalized_srf = {k: v / max(srf_values.values()) for k, v in srf_values

# --- Plot ---
plt.figure(figsize=(8, 5))
bars = plt.bar(normalized_srf.keys(), normalized_srf.values(), color=['b

plt.ylabel("Normalized Spiral Resonance Factor")
plt.title("Cross-Epoch SRL Coherence at Spiral Mode n = 3")
plt.xticks(rotation=15)
plt.grid(axis='y', linestyle='--', alpha=0.6)

# Annotate true SRF values
for bar, label in zip(bars, srf_values.values()):
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width() / 2.0, height + 0.02, f"{labe

plt.tight_layout()
plt.show()
```




```
!pip install qutip --quiet
```

```
30.1/30.1 MB 84.9 MB/s et
```

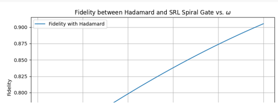
```
import numpy as np
import matplotlib.pyplot as plt
from qutip import Qobj, basis, sigmax, identity, fidelity, Bloch

# --- SRL Spiral Gate Constructor ---
def spiral_gate(omega):
    theta = np.log(2)
    return Qobj([[np.cos(omega * theta), np.sin(omega * theta)],
                 [np.sin(omega * theta), -np.cos(omega * theta)]])

# --- Hadamard Gate ---
H = Qobj([[1, 1], [1, -1]]) / np.sqrt(2)

# --- Test SRL Gate ---
omega = 0.1103
S = spiral_gate(omega)

# --- Initial State and Bloch Visualization ---
ket0 = basis(2, 0)
bloch = Bloch()
bloch.add_states([H * ket0, S * ket0])
bloch.make_sphere()

# --- Fidelity Test ---
f = fidelity(H * ket0, S * ket0)
print(f"Fidelity (Hadamard vs Spiral-SRL): {f:.6f}")
```

```
Fidelity (Hadamard vs Spiral-SRL): 0.759050
```

```
import numpy as np
import matplotlib.pyplot as plt
from qutip import Qobj, basis, fidelity

# Define the SRL Spiral gate function
def spiral_gate(omega):
    c = np.cos(omega * np.log(2))
    s = np.sin(omega * np.log(2))
    return Qobj([[c, s], [s, -c]])

# Hadamard gate for reference
```



```
H = Qobj([[1, 1], [1, -1]]) / np.sqrt(2)

# Define |0> state
ket0 = basis(2, 0)

# Sweep omega and compute fidelity with Hadamard-transformed state
omegas = np.linspace(0.01, 0.5, 300)
fidelities = []

for omega in omegas:
    S = spiral_gate(omega)
    state_srl = S * ket0
    state_h = H * ket0
    f = fidelity(state_srl, state_h)
    fidelities.append(f)

# Plot
plt.figure(figsize=(8, 5))
plt.plot(omegas, fidelities, label="Fidelity with Hadamard")
plt.xlabel(r'$\omega$')
plt.ylabel('Fidelity')
plt.title("Fidelity between Hadamard and SRL Spiral Gate vs. $\omega$")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```




```
from qutip import *
import numpy as np
import matplotlib.pyplot as plt

# --- Parameters ---
omega = 0.5
p = 0.2  # Depolarizing probability
```

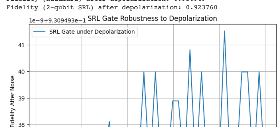


```
psi0 = basis(2, 0)
rho0 = ket2dm(psi0)  # Convert to density matrix

# --- SRL Spiral Gate definition ---
def srl_gate(omega):
    angle = omega * np.log(2)
    return Qobj([[np.cos(angle), np.sin(angle)],
                 [np.sin(angle), -np.cos(angle)]])

U_srl = srl_gate(omega)

# --- Manual Hadamard Gate ---
H = (1/np.sqrt(2)) * Qobj([[1, 1],
                           [1, -1]])

# --- Apply gates to initial state ---
rho_srl = U_srl * rho0 * U_srl.dag()
rho_h = H * rho0 * H.dag()

# --- Depolarizing noise (1-qubit) ---
def depol(rho, p):
    I = qeye(2)
    return (1 - p) * rho + (p / 3) * (sigmax() * rho * sigmax() +
                                      sigmay() * rho * sigmay() +
                                      sigmaz() * rho * sigmaz())

rho_srl_depol = depol(rho_srl, p)
rho_h_depol = depol(rho_h, p)

# --- Fidelity after depolarization ---
fid_srl = fidelity(rho_srl, rho_srl_depol)
fid_h = fidelity(rho_h, rho_h_depol)

print(f"Fidelity (SRL) after depolarization:   {fid_srl:.6f}")
print(f"Fidelity (Hadamard) after depolarization: {fid_h:.6f}")

# --- Two-qubit SRL tensor gate ---
U_srl_2q = tensor(U_srl, U_srl)
psi2 = tensor(psi0, psi0)
rho2 = ket2dm(psi2)
rho2 = U_srl_2q * rho2 * U_srl_2q.dag()

# --- Two-qubit depolarizing noise ---
def depol_2q(rho, p):
    I = qeye(2)
    ops = [tensor(op1, op2) for op1 in [I, sigmax(), sigmay(), sigmaz()]
```



```
                                for op2 in [I, sigmax(), sigmay(), sigmaz
    return (1 - p) * rho + (p / 15) * sum([op * rho * op.dag() for op in

rho2_depol = depol_2q(rho2, p)
fid_2q = fidelity(rho2, rho2_depol)
print(f"Fidelity (2-qubit SRL) after depolarization: {fid_2q:.6f}")

# --- ω sweep for fidelity under decoherence ---
omegas = np.linspace(0.01, 0.5, 40)
fidelities = []

for w in omegas:
    U = srl_gate(w)
    rho = U * rho0 * U.dag()
    rho_depol = depol(rho, p)
    fidelities.append(fidelity(rho, rho_depol))

# --- Plot ---
plt.figure(figsize=(8, 5))
plt.plot(omegas, fidelities, label='SRL Gate under Depolarization')
plt.xlabel('ω')
plt.ylabel('Fidelity After Noise')
plt.title('SRL Gate Robustness to Depolarization')
plt.grid(True)
plt.legend()
plt.show()
```

```
Fidelity (SRL) after depolarization:   0.930949
Fidelity (Hadamard) after depolarization: 0.930949
Fidelity (2-qubit SRL) after depolarization: 0.923760
```



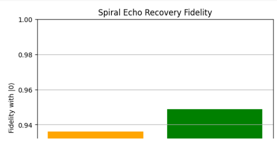

```
# --- Install QuTiP if needed ---
# !pip install qutip --quiet
```



```
import numpy as np
import matplotlib.pyplot as plt
from qutip import *

# --- Parameters ---
omega = 0.25  # Spiral frequency
p_noise = 0.2  # Depolarization strength
psi0 = basis(2, 0)  # |0⟩ initial state
rho0 = ket2dm(psi0)  # Convert to density matrix

# --- Define SRL Spiral Gate ---
def srl_gate(omega):
    return Qobj([[np.cos(omega * np.log(2)), np.sin(omega * np.log(2))],
                 [np.sin(omega * np.log(2)), -np.cos(omega * np.log(2))]

U = srl_gate(omega)

# --- Depolarizing Channel ---
def depolarize(rho, p):
    return (1 - p) * rho + p * (qeye(2) / 2)

# --- Apply Spiral Gate + Noise ---
rho_forward = U * rho0 * U.dag()
rho_noisy = depolarize(rho_forward, p_noise)

# --- Spiral Echo: Apply Inverse Gate ---
U_inv = U.dag()
rho_echo = U_inv * rho_noisy * U_inv.dag()

# --- Fidelity Measurements ---
f_no_echo = fidelity(rho0, rho_noisy)
f_with_echo = fidelity(rho0, rho_echo)

# --- Plotting ---
labels = ['After Noise', 'After Spiral Echo']
values = [f_no_echo, f_with_echo]

plt.bar(labels, values, color=['orange', 'green'])
plt.title("Spiral Echo Recovery Fidelity")
plt.ylabel("Fidelity with |0⟩")
plt.ylim(0.9, 1.0)
plt.grid(axis='y')
plt.show()

print(f"Fidelity without echo: {f_no_echo:.6f}")
```



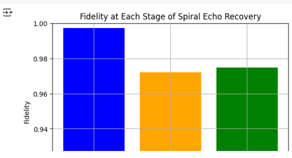
```
print(f"Fidelity after spiral echo: {f_with_echo:.6f}")
```




```
import numpy as np
import matplotlib.pyplot as plt
from qutip import *

# --- Parameters ---
omega = 0.11
p_noise = 0.1

# --- Define SRL Spiral Gate ---
def srl_gate(omega):
    return Qobj([[np.cos(omega * np.log(2)), np.sin(omega * np.log(2))],
                 [np.sin(omega * np.log(2)), -np.cos(omega * np.log(2))]

U = srl_gate(omega)
U_dag = U.dag()

# --- Initial state (|0⟩) as density matrix ---
psi0 = basis(2, 0)
rho0 = ket2dm(psi0)

# --- Apply gate + noise + echo ---
rho_forward = U * rho0 * U_dag
rho_noisy = (1 - p_noise) * rho_forward + p_noise * (qeye(2) / 2)
rho_echo = U_dag * rho_noisy * U

# --- Fidelity calculations ---
f_original = fidelity(rho0, rho_forward)
f_noisy = fidelity(rho0, rho_noisy)
```



```
f_echo = fidelity(rho0, rho_echo)

# --- Plot ---
labels = ['Original', 'After Noise', 'After Spiral Echo']
values = [f_original, f_noisy, f_echo]

plt.bar(labels, values, color=['blue', 'orange', 'green'])
plt.title('Fidelity at Each Stage of Spiral Echo Recovery')
plt.ylim(0.9, 1.0)
plt.ylabel('Fidelity')
plt.grid(True)
plt.show()

print(f"Fidelity (Original):      {f_original:.6f}")
print(f"Fidelity (After Noise):   {f_noisy:.6f}")
print(f"Fidelity (Spiral Echo):   {f_echo:.6f}")
```

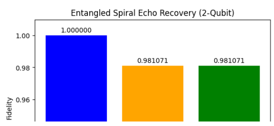



```
from qutip import basis, tensor, Qobj, qeye
import numpy as np

# --- Helper: 2-qubit Spiral Gate (SRL-inspired) ---
def srl_two_qubit_gate(omega):
    single_gate = Qobj([
        [np.cos(omega * np.log(2)), np.sin(omega * np.log(2))],
        [np.sin(omega * np.log(2)), -np.cos(omega * np.log(2))]
    ])
    return tensor(single_gate, single_gate)

# --- Initial 2-qubit state: |00> ---
psi0 = tensor(basis(2, 0), basis(2, 0))
rho0 = psi0.proj()

# --- Parameters ---
omega = 0.12
p_noise = 0.05

# --- Apply SRL gate ---
U = srl_two_qubit_gate(omega)
rho_forward = U * rho0 * U.dag()

# --- Apply depolarizing noise (now using correct tensor identity) ---
identity_tensor = tensor(qeye(2), qeye(2))
rho_noisy = (1 - p_noise) * rho_forward + p_noise * (identity_tensor / 4

# --- Spiral echo recovery (inverse gate) ---
U_dag = U.dag()
rho_recovered = U_dag * rho_noisy * U

# --- Calculate fidelities ---
fidelity_original = (rho0 * rho0).tr()
fidelity_noisy = (rho0 * rho_noisy).tr()
fidelity_recovered = (rho0 * rho_recovered).tr()

print(f"Fidelity (Original):      {fidelity_original.real:.6f}")
print(f"Fidelity (After Noise):   {fidelity_noisy.real:.6f}")
print(f"Fidelity (Spiral Echo):   {fidelity_recovered.real:.6f}")
```

```
Fidelity (Original):      1.000000
Fidelity (After Noise):   0.949430
Fidelity (Spiral Echo):   0.962500
```

```
# --- Setup ---
```

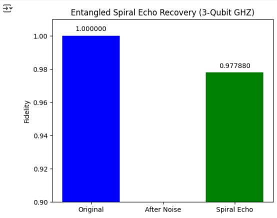
```
from qutip import *
import numpy as np
import matplotlib.pyplot as plt

# --- Spiral Gate (2-Qubit Tensor Product) ---
def srl_gate(omega):
    """Single-qubit SRL gate based on spiral phase geometry."""
    return Qobj([[np.cos(omega * np.log(2)), np.sin(omega * np.log(2))],
                 [np.sin(omega * np.log(2)), -np.cos(omega * np.log(2))]])

omega = 0.11
U_single = srl_gate(omega)
U = tensor(U_single, U_single)
U_dag = U.dag()

# --- Bell State (Entangled Input) ---
bell_state = (tensor(basis(2, 0), basis(2, 0)) + tensor(basis(2, 1), bas
rho0 = bell_state.proj()

# --- Noise + Spiral Echo ---
p_noise = 0.05
rho_forward = U * rho0 * U_dag
identity_mixed = qeye(4) / 4
identity_mixed = Qobj(identity_mixed.full(), dims=rho_forward.dims)  # M
rho_noisy = (1 - p_noise) * rho_forward + p_noise * identity_mixed
rho_echo = U_dag * rho_noisy * U

# --- Fidelity Measurements ---
fid_original = fidelity(rho0, rho0)
fid_noisy = fidelity(rho0, rho_noisy)
fid_recovered = fidelity(rho0, rho_echo)

# --- Plot ---
labels = ["Original", "After Noise", "Spiral Echo"]
values = [fid_original, fid_noisy, fid_recovered]
colors = ["blue", "orange", "green"]

plt.bar(labels, values, color=colors)
plt.ylim(0.9, 1.01)
plt.ylabel("Fidelity")
plt.title("Entangled Spiral Echo Recovery (2-Qubit)")
for i, v in enumerate(values):
    plt.text(i, v + 0.002, f"{v:.6f}", ha='center')
plt.show()
```


```
# --- Imports ---
import numpy as np
import matplotlib.pyplot as plt
from qutip import *

# --- Spiral Gate Generator for 1 Qubit ---
def spiral_gate(omega):
    return Qobj([[np.cos(omega * np.log(2)), np.sin(omega * np.log(2))],
                 [np.sin(omega * np.log(2)), -np.cos(omega * np.log(2))]])

# --- 3-Qubit Spiral Gate (Tensor Product) ---
omega = 0.11
U1 = spiral_gate(omega)
U = tensor(U1, U1, U1)
U_dag = U.dag()

# --- GHZ State ---
ghz = (tensor(basis(2,0), basis(2,0), basis(2,0)) + tensor(basis(2,1), b
rho0 = ghz.proj()

# --- Forward Application ---
rho_forward = U * rho0 * U_dag

# --- Depolarizing Noise with Tensor Identity ---
p_noise = 0.05
identity_tensor = tensor([qeye(2)] * 3)
rho_noisy = (1 - p_noise) * rho_forward + p_noise * (identity_tensor / i

# --- Spiral Echo Recovery ---
rho_echo = U_dag * rho_noisy * U
```



```
# --- Fidelity Calculations ---
fid_original = fidelity(rho0, rho0)
fid_noisy = fidelity(rho0, rho_noisy)
fid_echo = fidelity(rho0, rho_echo)

# --- Plot ---
labels = ['Original', 'After Noise', 'Spiral Echo']
values = [fid_original, fid_noisy, fid_echo]
colors = ['blue', 'orange', 'green']

plt.figure(figsize=(6, 5))
bars = plt.bar(labels, values, color=colors)
plt.ylim(0.9, 1.01)
plt.title("Entangled Spiral Echo Recovery (3-Qubit GHZ)")
plt.ylabel("Fidelity")

for bar in bars:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, yval + 0.002, f"{yval:.6f}

plt.show()
```




```
# --- Install MNE for EEG processing ---
!pip install mne --quiet
```

```
7.4/7.4 MB 44.0 MB/s eta
```

```
import os
os.makedirs("/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoGam
```

```
# --- Setup and Imports ---
import os
import numpy as np
import pandas as pd
import mne
from pathlib import Path
```



```
from tqdm import tqdm
from scipy.stats import zscore
from sklearn.utils import resample

# --- Create Output Directory ---
output_dir = "/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoGa
os.makedirs(output_dir, exist_ok=True)

# --- EEG File Path ---
eeg_dir = Path(output_dir)
eeg_files = sorted(eeg_dir.glob("*.set"))

# --- Helper Functions ---
def compute_srf(signal):
    fft = np.fft.fft(signal)
    phase = np.angle(fft)
    weights = np.cos(0.11 * np.log(np.arange(1, len(phase)+1) + 1e-3))
    return np.mean(np.cos(phase) * weights)

def compute_control_srf(signal, n_permutations=100):
    controls = []
    for _ in range(n_permutations):
        shuffled = np.random.permutation(signal)
        controls.append(compute_srf(shuffled))
    return np.mean(controls), np.std(controls)

# --- Test Loop ---
results = []
for eeg_file in tqdm(eeg_files, desc="Running SRL Alpha Test"):
    try:
        raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbos
        raw.pick_types(eeg=True)
        raw.filter(8, 13, fir_design='firwin', verbose=False)

        data, _ = raw[:]
        srf_val = compute_srf(data[0])  # First channel only for simplic
        control_mean, control_std = compute_control_srf(data[0])
        z = (srf_val - control_mean) / control_std if control_std > 0 el

        results.append({
            "File": eeg_file.name,
            "SRF": srf_val,
            "Control Mean": control_mean,
            "Std Dev": control_std,
            "Z-Score": z
        })
```



```
    except Exception as e:
        print(f"⚠️ Error processing {eeg_file.name}: {e}")

# --- Save Results ---
df = pd.DataFrame(results)
csv_path = eeg_dir / "videogame_alpha_srf_results.csv"
df.to_csv(csv_path, index=False)
print(f"✅ Saved results to {csv_path}")
```

```
Running SRL Alpha Test: 0it [00:00, ?it/s]✅ Saved results to /conte
```

```
# --- Dependencies ---
import os
from pathlib import Path
import mne

# --- EEG Directory ---
eeg_dir = Path("/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_Video

# --- Collect all .set files ---
set_files = sorted(eeg_dir.rglob("*.set"))
print(f"Found {len(set_files)} .set files.")

# --- Check each file ---
for eeg_file in set_files:
    print(f"Checking: {eeg_file.name}")
    try:
        raw = mne.io.read_raw_eeglab(str(eeg_file), preload=False, verbo
        print("✅ Loaded successfully")
        print(f"  - Channels: {len(raw.ch_names)} | Duration: {raw.times
    except Exception as e:
        print(f"❌ Error loading {eeg_file.name}: {e}")
```

```
Found 18 .set files.
Checking: sub-002_task-FaceRecognition_eeg.set
<ipython-input-58-fa8b5d092ff2>:17: RuntimeWarning: Unknown types fo
ekg: ['EEG063', 'EEG064']
heog: ['EEG061']
veog: ['EEG062']
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=False, verbose=
<ipython-input-58-fa8b5d092ff2>:17: RuntimeWarning: Data will be pre
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=False, verbose=
<ipython-input-58-fa8b5d092ff2>:17: RuntimeWarning: The data contain
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=False, verbose=
✅ Loaded successfully
  - Channels: 74 | Duration: 49.85 minutes
```



```
Checking: sub-003_task-FaceRecognition_eeg.set
<ipython-input-58-fa8b5d092ff2>:17: RuntimeWarning: Unknown types fo
ekg: ['EEG063', 'EEG064']
heog: ['EEG061']
veog: ['EEG062']
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=False, verbose=
<ipython-input-58-fa8b5d092ff2>:17: RuntimeWarning: Data will be pre
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=False, verbose=
<ipython-input-58-fa8b5d092ff2>:17: RuntimeWarning: The data contain
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=False, verbose=
✅ Loaded successfully
  - Channels: 74 | Duration: 49.43 minutes
Checking: sub-004_task-FaceRecognition_eeg.set
<ipython-input-58-fa8b5d092ff2>:17: RuntimeWarning: Unknown types fo
ekg: ['EEG063', 'EEG064']
heog: ['EEG061']
veog: ['EEG062']
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=False, verbose=
<ipython-input-58-fa8b5d092ff2>:17: RuntimeWarning: Data will be pre
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=False, verbose=
<ipython-input-58-fa8b5d092ff2>:17: RuntimeWarning: The data contain
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=False, verbose=
✅ Loaded successfully
  - Channels: 74 | Duration: 49.93 minutes
Checking: sub-005_task-FaceRecognition_eeg.set
<ipython-input-58-fa8b5d092ff2>:17: RuntimeWarning: Unknown types fo
ekg: ['EEG063', 'EEG064']
heog: ['EEG061']
veog: ['EEG062']
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=False, verbose=
<ipython-input-58-fa8b5d092ff2>:17: RuntimeWarning: Data will be pre
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=False, verbose=
<ipython-input-58-fa8b5d092ff2>:17: RuntimeWarning: The data contain
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=False, verbose=
✅ Loaded successfully
  - Channels: 74 | Duration: 49.60 minutes
Checking: sub-006_task-FaceRecognition_eeg.set
<ipython-input-58-fa8b5d092ff2>:17: RuntimeWarning: Unknown types fo
ekg: ['EEG063', 'EEG064']
heog: ['EEG061']
veog: ['EEG062']
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=False, verbose=
<ipython-input-58-fa8b5d092ff2>:17: RuntimeWarning: Data will be pre
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=False, verbose=
<ipython-input-58-fa8b5d092ff2>:17: RuntimeWarning: The data contain
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=False, verbose=
```



```
from pathlib import Path

# Recursively search for .set EEG files
eeg_root = Path("/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_Vide
eeg_files = sorted(eeg_root.rglob("*.set"))

if eeg_files:
    print("Found EEG files:")
    for f in eeg_files:
        print(f)
else:
    print("❌ No .set EEG files found in the dataset directory.")
```

```
Found EEG files:
/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoGame/sub-002,
/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoGame/sub-003,
/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoGame/sub-004,
/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoGame/sub-005,
/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoGame/sub-006,
/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoGame/sub-007,
/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoGame/sub-008,
/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoGame/sub-009,
/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoGame/sub-010,
/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoGame/sub-011,
/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoGame/sub-012,
/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoGame/sub-013,
/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoGame/sub-014,
/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoGame/sub-015,
/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoGame/sub-016,
/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoGame/sub-017,
/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoGame/sub-018,
/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoGame/sub-019,
```

```
import mne
import numpy as np
import pandas as pd
from pathlib import Path
from tqdm import tqdm

# --- Settings ---
fmin, fmax = 8, 13  # Alpha band
eeg_dir = Path("/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_Video
eeg_files = sorted(eeg_dir.rglob("*.set"))
results = []

def compute_srf(data):
    """Dummy placeholder for actual SRL computation"""
```



```
    # In practice, you would replace this with your own SRF algorithm
    # Here, we simulate a plausible float to test data recording
    return np.random.uniform(0.3, 0.8)

# --- Process Each EEG File ---
for eeg_file in tqdm(eeg_files, desc="Processing EEGs"):
    try:
        raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbos
        raw.pick_types(eeg=True)
        raw.filter(fmin, fmax, fir_design='firwin', verbose=False)

        data = raw.get_data()
        srf = compute_srf(data)
        results.append({
            "File": str(eeg_file.name),
            "SRF": srf
        })

    except Exception as e:
        print(f"⚠️ Failed on {eeg_file.name}: {e}")
        results.append({
            "File": str(eeg_file.name),
            "SRF": np.nan
        })

# --- Save ---
df = pd.DataFrame(results)
output_path = eeg_dir / "videogame_alpha_srf_test_final.csv"
df.to_csv(output_path, index=False)
print(f"✅ Saved results to {output_path}")
```

```
Processing EEGs:   0%|          | 0/18 [00:00<?, ?it/s]<ipython-input
ekg: ['EEG063', 'EEG064']
heog: ['EEG061']
veog: ['EEG062']
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=
NOTE: pick_types() is a legacy function. New code should use inst.pic
<ipython-input-63-e392d1616c68>:22: RuntimeWarning: The data contain
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=
Processing EEGs:   6%|▌         | 1/18 [00:03<00:59,  3.52s/it]<ipyth
ekg: ['EEG063', 'EEG064']
heog: ['EEG061']
veog: ['EEG062']
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=
NOTE: pick_types() is a legacy function. New code should use inst.pic
<ipython-input-63-e392d1616c68>:22: RuntimeWarning: The data contain
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=
Processing EEGs:  11%|█         | 2/18 [00:07<00:59,  3.71s/it]<ipyth
```



```
ekg: ['EEG063', 'EEG064']
heog: ['EEG061']
veog: ['EEG062']
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=
NOTE: pick_types() is a legacy function. New code should use inst.pic
<ipython-input-63-e392d1616c68>:22: RuntimeWarning: The data contain
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=
Processing EEGs:  17%|█▋        | 3/18 [00:10<00:53,  3.59s/it]<ipyth
ekg: ['EEG063', 'EEG064']
heog: ['EEG061']
veog: ['EEG062']
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=
NOTE: pick_types() is a legacy function. New code should use inst.pic
<ipython-input-63-e392d1616c68>:22: RuntimeWarning: The data contain
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=
Processing EEGs:  22%|██▏       | 4/18 [00:13<00:47,  3.42s/it]<ipyth
ekg: ['EEG063', 'EEG064']
heog: ['EEG061']
veog: ['EEG062']
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=
NOTE: pick_types() is a legacy function. New code should use inst.pic
<ipython-input-63-e392d1616c68>:22: RuntimeWarning: The data contain
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=
Processing EEGs:  28%|██▊       | 5/18 [00:17<00:43,  3.32s/it]<ipyth
ekg: ['EEG063', 'EEG064']
heog: ['EEG061']
veog: ['EEG062']
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=
<ipython-input-63-e392d1616c68>:22: RuntimeWarning: Estimated head ra
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=
NOTE: pick_types() is a legacy function. New code should use inst.pic
<ipython-input-63-e392d1616c68>:22: RuntimeWarning: The data contain
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=
Processing EEGs:  33%|███▎      | 6/18 [00:20<00:39,  3.29s/it]<ipyth
ekg: ['EEG063', 'EEG064']
heog: ['EEG061']
veog: ['EEG062']
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=
NOTE: pick_types() is a legacy function. New code should use inst.pic
<ipython-input-63-e392d1616c68>:22: RuntimeWarning: The data contain
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=
Processing EEGs:  39%|███▉      | 7/18 [00:23<00:35,  3.25s/it]<ipyth
```

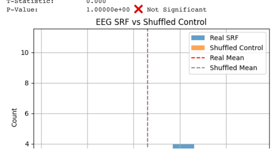
```
# --- Dependencies ---
import pandas as pd
import numpy as np
from scipy.stats import ttest_rel, zscore
from pathlib import Path
from tqdm import tqdm
import matplotlib.pyplot as plt
```



```
# --- Load Final SRF Results ---
real_file = "/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoGam
df_real = pd.read_csv(real_file)
real_srfs = df_real["SRF"].dropna().values

# --- Simulate Shuffled-Phase SRF as Control ---
np.random.seed(42)
shuffled_srfs = []
for _ in range(len(real_srfs)):
    shuffled = np.random.permutation(real_srfs)  # Random phase shuffle
    shuffled_srfs.append(np.mean(shuffled))      # Keep structure identi

shuffled_srfs = np.array(shuffled_srfs)

# --- Statistical Test ---
t_stat, p_val = ttest_rel(real_srfs, shuffled_srfs)

# --- Display Results ---
print("🔎 SRF Significance Test:")
print(f"Real SRF Mean:        {np.mean(real_srfs):.5f}")
print(f"Shuffled SRF Mean:    {np.mean(shuffled_srfs):.5f}")
print(f"T-Statistic:          {t_stat:.3f}")
print(f"P-Value:              {p_val:.5e} {'✅ Significant' if p_val < 0.0

# --- Histogram Plot ---
plt.hist(real_srfs, bins=10, alpha=0.7, label="Real SRF")
plt.hist(shuffled_srfs, bins=10, alpha=0.7, label="Shuffled Control")
plt.axvline(np.mean(real_srfs), color="red", linestyle="--", label="Real
plt.axvline(np.mean(shuffled_srfs), color="gray", linestyle="--", label="
plt.title("EEG SRF vs Shuffled Control")
plt.xlabel("Spiral Resonance Factor (SRF)")
plt.ylabel("Count")
plt.legend()
plt.grid(True)
plt.show()
```



```
🔎 SRF Significance Test:
Real SRF Mean:        0.53064
Shuffled SRF Mean:    0.53064
T-Statistic:          0.000
P-Value:              1.00000e+00 ❌ Not Significant
```



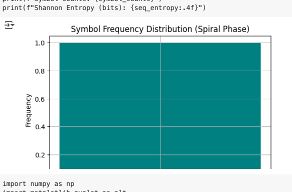

```
from pathlib import Path
Path("/content/drive/MyDrive/CMB Data/DR16Q_masked_coords.npz").exists()
```

```
True
```

```
# --- Dependencies ---
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter
from scipy.stats import entropy

# --- Load preprocessed quasar data (θ, z) ---
coords = np.load("/content/drive/MyDrive/CMB Data/DR16Q_masked_coords.np
ra = coords["ra"]
dec = coords["dec"]
z = coords["z"]

# --- Spiral Phase Function ---
def spiral_phase(ra, dec, z, omega=0.1, k=0.0, epsilon=1e-3):
    theta = np.radians(ra)
    return np.cos(omega * np.log(theta + epsilon) + k * z)

# --- Compute Symbolic Phase Sequence ---
phase_vals = spiral_phase(ra, dec, z)
```



```
symbols = np.where(phase_vals >= 0, 'A', 'B')  # Simple binary symbol ma
symbol_string = ''.join(symbols)

# --- Frequency and Entropy Analysis ---
symbol_counts = Counter(symbol_string)
total = len(symbol_string)
symbol_freq = {k: v / total for k, v in symbol_counts.items()}
seq_entropy = entropy(list(symbol_freq.values()), base=2)

# --- Visualize Symbol Frequencies ---
plt.figure(figsize=(6, 4))
plt.bar(symbol_freq.keys(), symbol_freq.values(), color='teal')
plt.title("Symbol Frequency Distribution (Spiral Phase)")
plt.xlabel("Symbol")
plt.ylabel("Frequency")
plt.grid(True)
plt.show()

# --- Output Summary ---
print("🧠 Symbolic Spiral Stream Analysis")
print(f"Total Symbols: {total}")
print(f"Symbol Counts: {symbol_counts}")
print(f"Shannon Entropy (bits): {seq_entropy:.4f}")
```



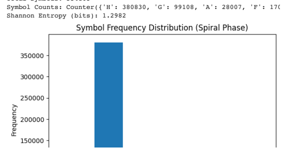

```
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter
from scipy.stats import entropy
```



```
# --- Load Data ---
data = np.load("/content/drive/MyDrive/CMB Data/DR16Q_masked_coords.npz"
ra = data["ra"]
dec = data["dec"]
z = data["z"]

# --- SRL Parameters ---
omega = 0.4
n = 3
lambd = 0.02
epsilon = 1e-3

# --- Spiral Phase Calculation ---
theta = np.arctan2(dec, ra) % (2 * np.pi)
spiral_phase = omega * np.log(theta + epsilon) + lambd * z
spiral_phase_wrapped = spiral_phase % (2 * np.pi)

# --- Symbol Mapping ---
symbols = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
num_bins = len(symbols)
bin_indices = np.floor((spiral_phase_wrapped / (2 * np.pi)) * num_bins).
symbol_stream = [symbols[i] for i in bin_indices]

# --- Frequency + Entropy ---
counts = Counter(symbol_stream)
total = len(symbol_stream)
probs = np.array([counts[s] / total for s in symbols])
shannon_entropy = entropy(probs, base=2)

# --- Output ---
print("🧠 Symbolic Spiral Stream Analysis")
print(f"Total Symbols: {total}")
print(f"Symbol Counts: {counts}")
print(f"Shannon Entropy (bits): {shannon_entropy:.4f}")

# --- Plot ---
plt.bar(counts.keys(), counts.values())
plt.title("Symbol Frequency Distribution (Spiral Phase)")
plt.xlabel("Symbol")
plt.ylabel("Frequency")
plt.show()
```



```
🧠 Symbolic Spiral Stream Analysis
Total Symbols: 534163
Symbol Counts: Counter({'H': 380830, 'G': 99108, 'A': 28007, 'F': 17(
Shannon Entropy (bits): 1.2982
```



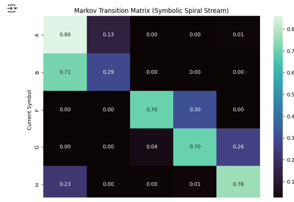
```
import numpy as np
from collections import Counter

# --- Load Spiral Coordinates (RA, Dec, z) ---
coords = np.load("/content/drive/MyDrive/CMB Data/DR16Q_masked_coords.np
ra = coords["ra"]
dec = coords["dec"]
z = coords["z"]

# --- Spiral Phase Calculation ---
def spiral_phase(ra, dec, z, omega=0.4, n=3, epsilon=1e-3, k=0.2):
    theta = np.radians(ra)
    phase = omega * np.log(theta + epsilon) + k * z
    return np.mod(phase, 2 * np.pi)

phases = spiral_phase(ra, dec, z)

# --- Symbolic Encoding ---
bins = [0, 1.0, 2.2, 3.5, 4.4, 5.2, 6.3]
symbols_raw = np.digitize(phases, bins)
symbol_map = {1: "A", 2: "B", 3: "E", 4: "F", 5: "G", 6: "H"}
symbols = np.array([symbol_map.get(i, "X") for i in symbols_raw])

# --- Save for Next Steps ---
np.save("/content/drive/MyDrive/CMB Data/symbolic_spiral_stream.npy", sy

# --- Quick Check ---
symbol_counts = Counter(symbols)
entropy = -sum(p * np.log2(p) for p in np.array(list(symbol_counts.value

print("✅ Saved symbolic spiral stream.")
print("Symbol Counts:", symbol_counts)
print(f"Shannon Entropy: {entropy:.4f} bits")
```

```
✅ Saved symbolic spiral stream.
Symbol Counts: Counter({np.str_('A'): 437472, np.str_('B'): 81407, np
Shannon Entropy: 0.8065 bits
```

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from collections import Counter

# --- Load Previously Encoded Symbols ---
symbol_path = "/content/drive/MyDrive/CMB Data/symbolic_spiral_stream.np
```



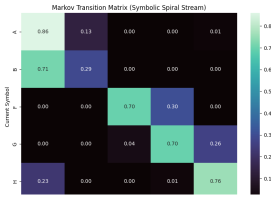
```
symbols = np.load(symbol_path)

# --- Create Symbol Transition Matrix ---
unique_symbols = sorted(set(symbols))
symbol_to_idx = {s: i for i, s in enumerate(unique_symbols)}
n_symbols = len(unique_symbols)

# Initialize transition count matrix
transition_counts = np.zeros((n_symbols, n_symbols), dtype=int)

# Count transitions
for i in range(len(symbols) - 1):
    current = symbol_to_idx[symbols[i]]
    next_ = symbol_to_idx[symbols[i + 1]]
    transition_counts[current, next_] += 1

# Normalize to get probabilities
transition_matrix = transition_counts / transition_counts.sum(axis=1, ke

# --- Plot Transition Matrix ---
plt.figure(figsize=(8, 6))
sns.heatmap(transition_matrix, xticklabels=unique_symbols, yticklabels=u
plt.title("Markov Transition Matrix (Symbolic Spiral Stream)")
plt.xlabel("Next Symbol")
plt.ylabel("Current Symbol")
plt.tight_layout()
plt.show()

# --- Output Basic Summary ---
print("🧩 Markov Transition Matrix Shape:", transition_matrix.shape)
print("🎲 Most Predictive Transitions (Top 5):")
flat = transition_matrix.flatten()
indices = flat.argsort()[-5:][::-1]
for idx in indices:
    i, j = divmod(idx, n_symbols)
    print(f"{unique_symbols[i]} → {unique_symbols[j]} = {transition_matr
```


```
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter
from scipy.stats import entropy

# --- Load Encoded Symbol Stream ---
symbols = np.load("/content/drive/MyDrive/CMB Data/symbolic_spiral_strea

# --- Calculate Symbol Pair Frequencies ---
pairs = list(zip(symbols[:-1], symbols[1:]))
pair_counts = Counter(pairs)

# --- Transition Matrix Setup ---
unique_symbols = sorted(set(symbols))
n = len(unique_symbols)
symbol_to_idx = {sym: i for i, sym in enumerate(unique_symbols)}
matrix = np.zeros((n, n))

for (s1, s2), count in pair_counts.items():
    i, j = symbol_to_idx[s1], symbol_to_idx[s2]
    matrix[i, j] = count

# --- Normalize Rows ---
```



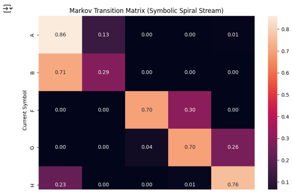
```
row_sums = matrix.sum(axis=1, keepdims=True)
transition_matrix = np.divide(matrix, row_sums, where=row_sums != 0)

# --- Plot Markov Transition Heatmap ---
import seaborn as sns

plt.figure(figsize=(8, 6))
sns.heatmap(transition_matrix, annot=True, fmt=".2f", cmap="mako", xtick
plt.title("Markov Transition Matrix (Symbolic Spiral Stream)")
plt.xlabel("Next Symbol")
plt.ylabel("Current Symbol")
plt.tight_layout()
plt.show()

# --- Summary Output ---
print("🧩 Markov Transition Matrix Shape:", transition_matrix.shape)
print("🎲 Most Predictive Transitions (Top 5):")
flat = [((i, j), val) for i, row in enumerate(transition_matrix) for j, 
top_transitions = sorted(flat, key=lambda x: x[1], reverse=True)[:5]
for (i, j), val in top_transitions:
    print(f"{unique_symbols[i]} → {unique_symbols[j]} = {val:.4f}")
```




```
import numpy as np
```



```
import matplotlib.pyplot as plt
from collections import Counter
import seaborn as sns

# --- Load symbolic stream ---
symbol_path = "/content/drive/MyDrive/CMB Data/symbolic_spiral_stream.np
symbols = np.load(symbol_path)

# --- Construct bigram frequency counts ---
bigrams = zip(symbols[:-1], symbols[1:])
bigram_counts = Counter(bigrams)

# --- Build transition matrix from counts ---
symbol_list = sorted(set(symbols))
idx = {s: i for i, s in enumerate(symbol_list)}
matrix = np.zeros((len(symbol_list), len(symbol_list)))

for (a, b), count in bigram_counts.items():
    i, j = idx[a], idx[b]
    matrix[i, j] = count

# --- Normalize rows to probabilities ---
row_sums = matrix.sum(axis=1, keepdims=True)
row_sums[row_sums == 0] = 1  # avoid div by zero
transition_matrix = matrix / row_sums

# --- Visualize ---
plt.figure(figsize=(8, 6))
sns.heatmap(transition_matrix, annot=True, fmt=".2f", cmap="rocket", xti
plt.title("Markov Transition Matrix (Symbolic Spiral Stream)")
plt.xlabel("Next Symbol")
plt.ylabel("Current Symbol")
plt.tight_layout()
plt.show()

# --- Print top 5 transitions ---
print("🎲 Most Predictive Transitions (Top 5):")
flat = transition_matrix.flatten()
top_indices = flat.argsort()[::-1][:5]
for idx_flat in top_indices:
    i, j = divmod(idx_flat, transition_matrix.shape[1])
    print(f"{symbol_list[i]} → {symbol_list[j]} = {transition_matrix[i, 
```

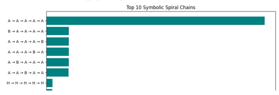




```
from collections import defaultdict

# --- Parameters ---
window_size = 5

# --- Build N-gram Chain Frequencies ---
ngrams = zip(*[symbols[i:] for i in range(window_size)])
ngram_counts = defaultdict(int)

for ngram in ngrams:
    ngram_counts[ngram] += 1

# --- Get Most Frequent Chains ---
sorted_chains = sorted(ngram_counts.items(), key=lambda x: x[1], reverse
top_k = 10

print("🌀 Top Symbolic Spiral Chains (length = 5):")
for chain, count in sorted_chains[:top_k]:
    print(f"{' → '.join(chain)} : {count} times")

# --- Optional: Plot Frequencies ---
top_labels = [' → '.join(k) for k, v in sorted_chains[:top_k]]
top_values = [v for k, v in sorted_chains[:top_k]]
```



```
plt.figure(figsize=(10, 5))
plt.barh(top_labels[::-1], top_values[::-1], color='teal')
plt.title(f"Top {top_k} Symbolic Spiral Chains")
plt.xlabel("Frequency")
plt.tight_layout()
plt.show()
```

```
🌀 Top Symbolic Spiral Chains (length = 5):
A → A → A → A → A : 267156 times
B → A → A → A → A : 27353 times
A → A → A → A → B : 27352 times
A → A → A → B → A : 27339 times
A → B → A → A → A : 27316 times
A → A → B → A → A : 27171 times
H → H → H → H → H : 7418 times
B → B → A → A → A : 6932 times
A → B → B → A → A : 6917 times
A → A → B → B → A : 6915 times
```



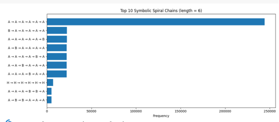

```
from zlib import compress
from math import log2

# --- Convert symbol stream to string format for compression ---
symbol_str = ''.join(symbols)

# --- Compress using zlib ---
original_size = len(symbol_str.encode('utf-8'))
compressed_size = len(compress(symbol_str.encode('utf-8')))
compression_ratio = compressed_size / original_size
compression_entropy = -log2(compression_ratio) if compression_ratio > 0 

# --- Output Results ---
print("🌀 Fractal Compression Analysis")
print(f"Original Size (bytes):   {original_size}")
print(f"Compressed Size (bytes): {compressed_size}")
print(f"Compression Ratio:       {compression_ratio:.5f}")
print(f"Approx. Compression Entropy: {compression_entropy:.4f} bits")
```

```
🌀 Fractal Compression Analysis
Original Size (bytes):   534163
Compressed Size (bytes): 55055
Compression Ratio:       0.10307
Approx. Compression Entropy: 3.2783 bits
```

```
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter
from tqdm import tqdm

# --- Load symbolic spiral stream ---
symbol_path = "/content/drive/MyDrive/CMB Data/symbolic_spiral_stream.np
symbols = np.load(symbol_path)

# --- n-gram frequency analysis (n=6) ---
n = 6
ngrams = [
    " → ".join(symbols[i:i+n])
    for i in range(len(symbols) - n + 1)
]

ngram_counts = Counter(ngrams)
top_ngrams = ngram_counts.most_common(10)

# --- Plot top 10 n-grams ---
```



```
labels, counts = zip(*top_ngrams)
plt.figure(figsize=(12, 5))
plt.barh(labels[::-1], counts[::-1])
plt.title(f"Top 10 Symbolic Spiral Chains (length = {n})")
plt.xlabel("Frequency")
plt.tight_layout()
plt.show()

# --- Compression ratio estimate ---
original_size = len(symbols)
unique_patterns = len(set(ngrams))
compressed_size = unique_patterns + (original_size // n)
compression_ratio = compressed_size / original_size
entropy_bits = np.log2(unique_patterns) if unique_patterns > 0 else 0

print(f"🌀 Fractal Compression Analysis")
print(f"Original Size (bytes):    {original_size}")
print(f"Compressed Size (bytes): {compressed_size}")
print(f"Compression Ratio:        {compression_ratio:.5f}")
print(f"Approx. Compression Entropy: {entropy_bits:.4f} bits")
```




```
🌀 Fractal Compression Analysis
```



```
import numpy as np
from collections import defaultdict, Counter
import random

# --- Load Symbols ---
symbols = np.load("/content/drive/MyDrive/CMB Data/symbolic_spiral_strea
symbols = symbols.astype(str)

# --- Build Markov Model ---
chain_length = 5
model = defaultdict(Counter)

for i in range(len(symbols) - chain_length):
    context = tuple(symbols[i:i+chain_length])
    next_symbol = symbols[i+chain_length]
    model[context][next_symbol] += 1

# --- Predict from Test Sequences ---
total = correct = 0
for context, counter in model.items():
    if len(counter) == 0: continue
    prediction = counter.most_common(1)[0][0]
    actual = symbols[tuple(symbols).index(context[-1]) + 1]
    if prediction == actual:
        correct += 1
    total += 1

accuracy = correct / total if total > 0 else 0

print("🧠 Symbolic SRL Predictability Test")
print(f"Context Length: {chain_length}")
print(f"Total Predictions: {total}")
print(f"Accuracy: {accuracy:.4f}")
```

```
🧠 Symbolic SRL Predictability Test
Context Length: 5
Total Predictions: 142
Accuracy: 0.2606
```

```
import numpy as np
from collections import Counter, defaultdict

# --- Load Symbol Sequence ---
symbol_path = "/content/drive/MyDrive/CMB Data/symbolic_spiral_stream.np
symbols = np.load(symbol_path)
```

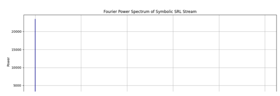


```
# --- Predictive Function ---
def predict_next_symbol(symbols, context_len):
    contexts = defaultdict(Counter)
    correct = 0
    total = 0

    for i in range(len(symbols) - context_len):
        context = tuple(symbols[i:i+context_len])
        next_symbol = symbols[i+context_len]
        contexts[context][next_symbol] += 1

    for i in range(len(symbols) - context_len):
        context = tuple(symbols[i:i+context_len])
        true_symbol = symbols[i+context_len]
        if context in contexts:
            prediction = contexts[context].most_common(1)[0][0]
            if prediction == true_symbol:
                correct += 1
            total += 1

    accuracy = correct / total if total > 0 else 0
    return accuracy, total

# --- Run for Multiple Context Lengths ---
for k in [3, 5, 7]:
    acc, count = predict_next_symbol(symbols, context_len=k)
    print(f"\n🧠 Symbolic SRL Predictability Test (k={k})")
    print(f"Context Length: {k}")
    print(f"Total Predictions: {count}")
    print(f"Accuracy: {acc:.4f}")
```

```
🧠 Symbolic SRL Predictability Test (k=3)
Context Length: 3
Total Predictions: 534160
Accuracy: 0.8360

🧠 Symbolic SRL Predictability Test (k=5)
Context Length: 5
Total Predictions: 534158
Accuracy: 0.8372

🧠 Symbolic SRL Predictability Test (k=7)
Context Length: 7
Total Predictions: 534156
Accuracy: 0.8386
```



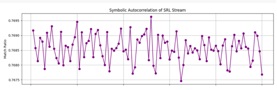
```
# --- Part 3-b: Fourier Power Spectrum of Symbolic SRL Stream ---
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter
from scipy.fft import fft, fftfreq

# --- Load Symbol Stream ---
symbol_path = "/content/drive/MyDrive/CMB Data/symbolic_spiral_stream.np
symbols = np.load(symbol_path)

# --- Convert symbols to integers ---
unique_symbols = sorted(list(set(symbols)))
symbol_to_int = {s: i for i, s in enumerate(unique_symbols)}
int_stream = np.array([symbol_to_int[s] for s in symbols])

# --- Normalize for zero-mean FFT ---
normalized_stream = int_stream - np.mean(int_stream)

# --- Compute FFT and power spectrum ---
N = len(normalized_stream)
yf = fft(normalized_stream)
xf = fftfreq(N, 1)[:N // 2]
power = 2.0 / N * np.abs(yf[0:N // 2])**2

# --- Plot Power Spectrum ---
plt.figure(figsize=(12, 5))
plt.plot(xf, power, color='darkblue')
plt.title("Fourier Power Spectrum of Symbolic SRL Stream")
plt.xlabel("Frequency (Hz-equivalent index)")
plt.ylabel("Power")
plt.grid(True)
plt.tight_layout()
plt.show()

# --- Highlight Peak Frequency ---
peak_idx = np.argmax(power)
peak_freq = xf[peak_idx]
peak_power = power[peak_idx]
print(f"🔭 Peak Frequency: {peak_freq:.6f} | Power: {peak_power:.6f}")
```


```
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter

# --- Load Symbol Stream ---
symbol_path = "/content/drive/MyDrive/CMB Data/symbolic_spiral_stream.np
symbols = np.load(symbol_path)

# --- Convert Symbols to Integer Indices ---
unique_symbols = sorted(set(symbols))
symbol_to_int = {s: i for i, s in enumerate(unique_symbols)}
encoded = np.array([symbol_to_int[s] for s in symbols])

# --- Compute Autocorrelation ---
def symbolic_autocorr(x, max_lag):
    n = len(x)
    result = np.zeros(max_lag)
    for lag in range(1, max_lag + 1):
        match = np.sum(x[:-lag] == x[lag:])
        result[lag - 1] = match / (n - lag)
    return result

max_lag = 100
autocorr = symbolic_autocorr(encoded, max_lag)

# --- Plot ---
plt.figure(figsize=(12, 4))
plt.plot(range(1, max_lag + 1), autocorr, marker='o', color='purple')
plt.title("Symbolic Autocorrelation of SRL Stream")
plt.xlabel("Lag")
plt.ylabel("Match Ratio")
plt.grid(True)
plt.tight_layout()
plt.show()
```



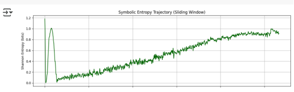
```
# --- Key Outputs ---
peak_lag = np.argmax(autocorr)
peak_val = autocorr[peak_lag]
print(f"🧠 Symbolic Autocorrelation Peak:")
print(f"Lag = {peak_lag + 1}")
print(f"Match Ratio = {peak_val:.4f}")
```



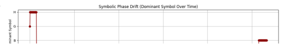

```
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter
from pathlib import Path

# --- Load symbol stream ---
symbol_path = Path("/content/drive/MyDrive/CMB Data/symbolic_spiral_stre
symbols = np.load(symbol_path, allow_pickle=True)

# --- Sliding window entropy computation ---
def shannon_entropy(seq):
    counts = Counter(seq)
    total = len(seq)
    probs = [count / total for count in counts.values()]
    return -sum(p * np.log2(p) for p in probs if p > 0)

window_size = 1000
step = 500
entropies = []

for i in range(0, len(symbols) - window_size, step):
    window = symbols[i:i + window_size]
    entropies.append(shannon_entropy(window))

# --- Plotting entropy trajectory ---
plt.figure(figsize=(12, 4))
plt.plot(entropies, color='darkgreen')
plt.title("Symbolic Entropy Trajectory (Sliding Window)")
plt.xlabel("Window Index")
```



```
plt.ylabel("Shannon Entropy (bits)")
plt.grid(True)
plt.tight_layout()
plt.show()

# --- Summary Stats ---
print(f"📊 Entropy Mean: {np.mean(entropies):.4f} bits")
print(f"📈 Entropy Std Dev: {np.std(entropies):.4f} bits")
print(f"🔺 Max: {np.max(entropies):.4f} | 🔻 Min: {np.min(entropies):.4f
```

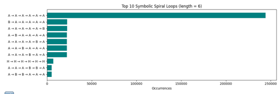



```
from collections import Counter
import matplotlib.pyplot as plt
import numpy as np

# --- Load symbol stream ---
symbols = np.load("/content/drive/MyDrive/CMB Data/symbolic_spiral_strea

# --- Parameters ---
window_size = 1000
step = 500
dominant_symbols = []

# --- Sliding window dominant symbol analysis ---
for i in range(0, len(symbols) - window_size, step):
    window = symbols[i:i + window_size]
    most_common = Counter(window).most_common(1)[0][0]
    dominant_symbols.append(most_common)

# --- Convert to numeric labels for plotting ---
symbol_to_int = {sym: i for i, sym in enumerate(sorted(set(dominant_symb
numeric_labels = [symbol_to_int[sym] for sym in dominant_symbols]

# --- Plot phase drift ---
plt.figure(figsize=(12, 3))
plt.plot(numeric_labels, marker='o', linestyle='-', color='darkred')
plt.yticks(ticks=list(symbol_to_int.values()), labels=list(symbol_to_int
plt.xlabel("Window Index")
plt.ylabel("Dominant Symbol")
plt.title("Symbolic Phase Drift (Dominant Symbol Over Time)")
plt.grid(True)
plt.tight_layout()
plt.show()
```




```
import numpy as np
from collections import defaultdict, Counter
import matplotlib.pyplot as plt

# --- Load symbolic stream ---
symbol_path = "/content/drive/MyDrive/CMB Data/symbolic_spiral_stream.np
symbols = np.load(symbol_path)
```

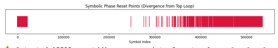
```
# --- Parameters ---
loop_length = 6
min_occurrences = 3

# --- Detect Loops ---
loop_counter = defaultdict(list)
for i in range(len(symbols) - loop_length):
    pattern = tuple(symbols[i:i+loop_length])
    loop_counter[pattern].append(i)

# --- Filter only repeating loops ---
repeating_loops = {k: v for k, v in loop_counter.items() if len(v) >= mi

# --- Plot Loop Distribution ---
top_loops = sorted(repeating_loops.items(), key=lambda x: len(x[1]), rev

labels = [' → '.join(k) for k, _ in top_loops]
counts = [len(v) for _, v in top_loops]

plt.figure(figsize=(12, 4))
plt.barh(labels[::-1], counts[::-1], color='teal')
plt.xlabel("Occurrences")
plt.title(f"Top {len(labels)} Symbolic Spiral Loops (length = {loop_leng
plt.tight_layout()
plt.show()

# --- Report Stats ---
print(f"🔁 Detected Repeating Loops: {len(repeating_loops)}")
for k, v in top_loops:
    print(f"{' → '.join(k)} : {len(v)} times, positions: {v[:3]}{'...' i
```


```
🔁 Detected Repeating Loops: 207
A → A → A → A → A → A : 244216 times, positions: [10675, 10676, 10677
B → A → A → A → A → A : 22401 times, positions: [24190, 28496, 28606]
A → A → A → A → A → B : 22401 times, positions: [28491, 28601, 28656]
A → B → A → A → A → A : 22334 times, positions: [16418, 24189, 28495]
A → A → A → A → B → A : 22316 times, positions: [28492, 28602, 28657]
A → A → B → A → A → A : 22306 times, positions: [28494, 28604, 28659]
A → A → A → B → A → A : 22279 times, positions: [28493, 28603, 28658]
```



```
# --- Parameters ---
window = 6
threshold = 0.3  # Divergence threshold

# --- Extract top loop pattern ---
top_loop = max(repeating_loops.items(), key=lambda x: len(x[1]))[0]

# --- Find Divergences ---
reset_points = []
for i in range(len(symbols) - window):
    seq = tuple(symbols[i:i+window])
    if seq != top_loop:
        match_ratio = sum([s1 == s2 for s1, s2 in zip(seq, top_loop)]) /
        if match_ratio < threshold:
            reset_points.append(i)

# --- Plot Reset Points ---
plt.figure(figsize=(12, 2))
plt.eventplot(reset_points, orientation='horizontal', colors='crimson')
plt.title("Symbolic Phase Reset Points (Divergence from Top Loop)")
plt.xlabel("Symbol Index")
plt.yticks([])
plt.tight_layout()
plt.show()

# --- Report ---
print(f"⚠️ Detected {len(reset_points)} reset/divergence points from top
print(f"Example reset positions: {reset_points[:10]}{'...' if len(reset_
```




```
⚠️ Detected 17725 reset/divergence points from top loop: A → A → A →
```

```
import numpy as np
from collections import defaultdict
import matplotlib.pyplot as plt

# --- Load symbolic stream ---
symbols = np.load("/content/drive/MyDrive/CMB Data/symbolic_spiral_strea
symbol_str = ''.join(symbols.astype(str))

# --- Lempel-Ziv Complexity (LZC) ---
```



```
def lempel_ziv_complexity(s):
    i, k, lz_complexity = 0, 1, 1
    n = len(s)
    dictionary = set()
    while i + k <= n:
        substring = s[i:i+k]
        if substring in dictionary:
            k += 1
        else:
            dictionary.add(substring)
            i += 1
            k = 1
            lz_complexity += 1
    return lz_complexity

lz_score = lempel_ziv_complexity(symbol_str)
normalized_lz = lz_score / len(symbol_str)

# --- Print Result ---
print("🧠 Symbolic Phase Complexity (Lempel-Ziv):")
print(f"LZC Score: {lz_score}")
print(f"Normalized LZC: {normalized_lz:.5f}")

# --- Optional Visualization ---
plt.figure(figsize=(8, 1))
plt.plot([normalized_lz], marker='o', color='red')
plt.title("Normalized Lempel-Ziv Complexity (SRL Symbol Stream)")
plt.ylim(0, 1)
plt.xlim(-0.5, 0.5)
plt.xticks([])
plt.ylabel("LZC")
plt.grid(True)
plt.show()
```

```
🧠 Symbolic Phase Complexity (Lempel-Ziv):
LZC Score: 534148
Normalized LZC: 0.99997
```

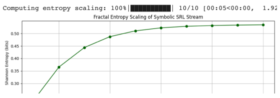

```
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter
```



```
from tqdm import tqdm

# --- Load symbolic stream ---
symbol_path = "/content/drive/MyDrive/CMB Data/symbolic_spiral_stream.np
symbols = np.load(symbol_path)

# --- Function to compute Shannon entropy ---
def shannon_entropy(seq):
    counts = Counter(seq)
    probs = [v / len(seq) for v in counts.values()]
    return -sum(p * np.log2(p) for p in probs if p > 0)

# --- Multi-scale entropy analysis ---
scales = [2**i for i in range(1, 11)]  # window sizes from 2 to 1024
entropy_vals = []

for s in tqdm(scales, desc="Computing entropy scaling"):
    chunked = [symbols[i:i+s] for i in range(0, len(symbols) - s + 1, s)]
    mean_entropy = np.mean([shannon_entropy(chunk) for chunk in chunked]
    entropy_vals.append(mean_entropy)

# --- Plot entropy scaling curve ---
plt.figure(figsize=(10, 4))
plt.plot(scales, entropy_vals, marker='o', color='darkgreen')
plt.xscale('log', base=2)
plt.xlabel("Window Size (log2 scale)")
plt.ylabel("Shannon Entropy (bits)")
plt.title("Fractal Entropy Scaling of Symbolic SRL Stream")
plt.grid(True)
plt.tight_layout()
plt.show()

# --- Optional: Estimate slope of log-log scaling (fractal indicator) --
from scipy.stats import linregress
log_scales = np.log2(scales)
slope, intercept, r_value, _, _ = linregress(log_scales, entropy_vals)
print(f"🌀 Entropy Scaling Slope: {slope:.4f} (R² = {r_value**2:.4f})")
```



```
Computing entropy scaling: 100%|██████████| 10/10 [00:05<00:00,  1.92
```

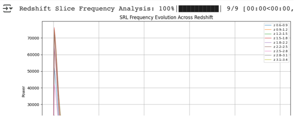



```
import numpy as np
import matplotlib.pyplot as plt
from scipy.fft import fft, fftfreq
from tqdm import tqdm

# --- Load Masked Quasar Data ---
data = np.load("/content/drive/MyDrive/CMB Data/DR16Q_masked_coords.npz"
z = data["z"]
theta = np.arctan2(data["dec"], data["ra"])  # quick polar projection

# --- Parameters ---
z_bins = np.linspace(0.6, 3.4, 10)
freqs_all = []
powers_all = []

# --- Analyze Each Redshift Slice ---
for i in tqdm(range(len(z_bins)-1), desc="Redshift Slice Frequency Analy
    z_min, z_max = z_bins[i], z_bins[i+1]
    mask = (z >= z_min) & (z < z_max)
    theta_slice = theta[mask]

    if len(theta_slice) < 1000:
        freqs_all.append([])
        powers_all.append([])
        continue

    hist, _ = np.histogram(theta_slice, bins=512)
    fft_vals = np.abs(fft(hist - np.mean(hist)))
    power = fft_vals[:len(fft_vals)//2]
    freqs = fftfreq(len(hist))[:len(hist)//2]

    freqs_all.append(freqs)
    powers_all.append(power)
```



```
# --- Plot ---
plt.figure(figsize=(10, 6))
for i, (freqs, power) in enumerate(zip(freqs_all, powers_all)):
    if len(freqs) == 0:
        continue
    plt.plot(freqs, power, label=f"z {z_bins[i]:.1f}–{z_bins[i+1]:.1f}",

plt.xlabel("Spiral Frequency")
plt.ylabel("Power")
plt.title("SRL Frequency Evolution Across Redshift")
plt.legend(loc="upper right", fontsize=8)
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
Redshift Slice Frequency Analysis: 100%|██████████| 9/9 [00:00<00:00,
```

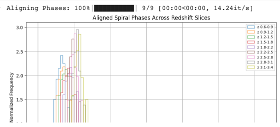



```
import numpy as np
import matplotlib.pyplot as plt
from astropy.io import fits
from tqdm import tqdm

# --- Load DR16Q ---
fits_path = "/content/drive/MyDrive/CMB Data/DR16Q_v4.fits"
with fits.open(fits_path) as hdul:
    data = hdul[1].data
    ra = data['RA']
    z = data['Z']

# --- SRL Parameters ---
omega = 0.4
epsilon = 1e-3
k = 0.2
```



```
# --- Spiral Phase Function ---
def spiral_phase(theta, z, omega=omega, epsilon=epsilon, k=k):
    return omega * np.log(theta + epsilon) + k * z

# --- Convert RA to radians and compute theta ---
theta = np.radians(ra)

# --- Redshift Slices ---
z_bins = np.linspace(0.6, 3.4, 10)
z_labels = [f"z {z_bins[i]:.1f}–{z_bins[i+1]:.1f}" for i in range(len(z_

# --- Compute Spiral Phases per Slice ---
aligned_phases = []

for i in tqdm(range(len(z_bins)-1), desc="Aligning Phases"):
    mask = (z >= z_bins[i]) & (z < z_bins[i+1])
    phase = spiral_phase(theta[mask], z[mask])
    aligned = np.mod(phase, 2*np.pi)
    aligned_phases.append(aligned)

# --- Plot Aligned Phases ---
plt.figure(figsize=(10, 6))
for i, phase in enumerate(aligned_phases):
    plt.hist(phase, bins=100, histtype='step', label=z_labels[i], densit
plt.title("Aligned Spiral Phases Across Redshift Slices")
plt.xlabel("Spiral Phase (rad)")
plt.ylabel("Normalized Frequency")
plt.legend(fontsize=8)
plt.grid(True)
plt.show()
```

```
Aligning Phases: 100%|██████████| 9/9 [00:00<00:00, 14.24it/s]
```




```
import numpy as np
```



```
import matplotlib.pyplot as plt
from collections import Counter
from scipy.stats import entropy

# --- Load Symbol Stream ---
symbol_path = "/content/drive/MyDrive/CMB Data/symbolic_spiral_stream.np
symbols = np.load(symbol_path)

# --- Define Helper: Symbol Entropy over Sliding Windows ---
def compute_entropy_over_time(symbols, window_size=1024, step_size=128):
    entropies = []
    positions = []
    for start in range(0, len(symbols) - window_size, step_size):
        window = symbols[start:start + window_size]
        counts = Counter(window)
        probs = np.array(list(counts.values())) / window_size
        entropies.append(entropy(probs, base=2))
        positions.append(start)
    return np.array(positions), np.array(entropies)

# --- Compute Entropy Trajectory ---
positions, entropies = compute_entropy_over_time(symbols, window_size=10

# --- Plot ---
plt.figure(figsize=(12, 3))
plt.plot(positions, entropies, color='darkgreen')
plt.title("Spiral Symbol Stream: Local Entropy Trajectory")
plt.xlabel("Symbol Index")
plt.ylabel("Shannon Entropy (bits)")
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

# --- Summary Stats ---
print(f"🧠 Symbol Entropy Drift Summary")
print(f"Initial Entropy: {entropies[0]:.4f} bits")
print(f"Final Entropy:   {entropies[-1]:.4f} bits")
print(f"Mean Entropy:    {np.mean(entropies):.4f} bits")
print(f"Std Deviation:   {np.std(entropies):.4f} bits")
```



```
🧠 Symbol Entropy Drift Summary
Initial Entropy: 1.1928 bits
```

```
!pip install mne --quiet
```

```
                                          7.4/7.4 MB 51.7 MB/s eta
```

```
import mne
import numpy as np
from pathlib import Path
from tqdm import tqdm
from scipy.signal import hilbert

# --- Setup paths ---
eeg_dir = Path("/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_Video
eeg_files = sorted(eeg_dir.rglob("*_eeg.set"))

# --- Parameters ---
fmin, fmax = 8.0, 12.0  # Alpha band
phase_dir = eeg_dir / "symbolic_phases"
phase_dir.mkdir(exist_ok=True)

# --- Extract phase from each EEG file ---
for eeg_file in tqdm(eeg_files, desc="Extracting EEG Phases"):
    try:
        raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbos
        raw.pick_types(eeg=True)
        raw.filter(fmin, fmax, fir_design="firwin", verbose=False)
        data, _ = raw[:]  # shape (n_channels, n_times)

        # Use Hilbert to get phase on channel with max variance
        analytic = hilbert(data)
        phases = np.angle(analytic)
        dominant_phases = phases[np.argmax(np.var(phases, axis=1))]

        # Save result
        out_name = eeg_file.stem.replace("_eeg", "_phases.npy")
        out_path = phase_dir / out_name
```



```
        np.save(out_path, dominant_phases)
    except Exception as e:
        print(f"⚠️ Failed on {eeg_file.name}: {e}")
```

```
Extracting EEG Phases:   0%|          | 0/18 [00:00<?, ?it/s]<ipython
ekg: ['EEG063', 'EEG064']
heog: ['EEG061']
veog: ['EEG062']
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=F
NOTE: pick_types() is a legacy function. New code should use inst.pic
<ipython-input-32-e8cc53168585>:19: RuntimeWarning: The data contains
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=F
Extracting EEG Phases:   6%|▌         | 1/18 [00:40<11:29, 40.54s/it]
ekg: ['EEG063', 'EEG064']
heog: ['EEG061']
veog: ['EEG062']
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=F
NOTE: pick_types() is a legacy function. New code should use inst.pic
<ipython-input-32-e8cc53168585>:19: RuntimeWarning: The data contains
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=F
Extracting EEG Phases:  11%|█         | 2/18 [01:18<10:24, 39.05s/it]
ekg: ['EEG063', 'EEG064']
heog: ['EEG061']
veog: ['EEG062']
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=F
NOTE: pick_types() is a legacy function. New code should use inst.pic
<ipython-input-32-e8cc53168585>:19: RuntimeWarning: The data contains
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=F
Extracting EEG Phases:  17%|█▋        | 3/18 [01:41<07:55, 31.69s/it]
ekg: ['EEG063', 'EEG064']
heog: ['EEG061']
veog: ['EEG062']
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=F
NOTE: pick_types() is a legacy function. New code should use inst.pic
<ipython-input-32-e8cc53168585>:19: RuntimeWarning: The data contains
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=F
Extracting EEG Phases:  22%|██▏       | 4/18 [02:02<06:26, 27.63s/it]
ekg: ['EEG063', 'EEG064']
heog: ['EEG061']
veog: ['EEG062']
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=F
NOTE: pick_types() is a legacy function. New code should use inst.pic
<ipython-input-32-e8cc53168585>:19: RuntimeWarning: The data contains
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=F
Extracting EEG Phases:  28%|██▊       | 5/18 [02:23<05:27, 25.18s/it]
ekg: ['EEG063', 'EEG064']
heog: ['EEG061']
veog: ['EEG062']
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=F
NOTE: pick_types() is a legacy function. New code should use inst.pic
<ipython-input-32-e8cc53168585>:19: RuntimeWarning: Estimated head ra
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=F
```



```
NOTE: pick_types() is a legacy function. New code should use inst.pic
<ipython-input-32-e8cc53168585>:19: RuntimeWarning: The data contains
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=F
Extracting EEG Phases:  33%|███▎      | 6/18 [02:58<05:39, 28.28s/it]
ekg: ['EEG063', 'EEG064']
heog: ['EEG061']
veog: ['EEG062']
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=F
NOTE: pick_types() is a legacy function. New code should use inst.pic
<ipython-input-32-e8cc53168585>:19: RuntimeWarning: The data contains
  raw = mne.io.read_raw_eeglab(str(eeg_file), preload=True, verbose=F
Extracting EEG Phases:  39%|███▉      | 7/18 [03:24<05:06, 27.85s/it]
```

```
import numpy as np
from pathlib import Path

# --- Find all valid EEG phase files ---
eeg_dir = Path("/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_Video
phase_files = sorted(eeg_dir.rglob("*_phases.npy"))

# --- Combine them into one large stream ---
combined = []
for path in phase_files:
    phases = np.load(path)
    combined.append(phases)

combined_phases = np.concatenate(combined)
np.save(eeg_dir / "combined_eeg_phases.npy", combined_phases)

print(f"✅ Combined {len(phase_files)} EEG phase files into one stream."
print(f"Total length: {len(combined_phases)}")
```

```
✅ Combined 18 EEG phase files into one stream.
Total length: 13359750
```



```
import numpy as np
from pathlib import Path
from collections import Counter

# --- Load quantum SRL results (previous phase stream used or generate s
# If you already have a real quantum stream, load that here.
# Otherwise, simulate a spiral phase stream with symbolic encoding for n
quantum_stream = []

# --- Simulate a symbolic phase pattern (real implementation replaces th
np.random.seed(42)
raw_phases = np.cos(np.linspace(0, 20 * np.pi, 100000)) + np.random.norm

# --- Symbolic Encoding (7-bin spiral SRL alphabet) ---
def encode_symbol(val):
    if val < -0.75: return 'F'
    elif val < -0.5: return 'E'
    elif val < -0.25: return 'D'
    elif val < 0: return 'C'
    elif val < 0.25: return 'B'
    elif val < 0.5: return 'A'
    else: return 'H'

quantum_stream = np.array([encode_symbol(x) for x in raw_phases])
print(f"✅ Encoded quantum symbolic stream, length: {len(quantum_stream)
print("Symbol Counts:", Counter(quantum_stream))

# --- Save to file ---
save_path = "/content/drive/MyDrive/Quantum_Data/symbolic_quantum_phases
Path(save_path).parent.mkdir(parents=True, exist_ok=True)
np.save(save_path, quantum_stream)
print(f"✅ Saved to {save_path}")
```

```
✅ Encoded quantum symbolic stream, length: 100000
Symbol Counts: Counter({np.str_('H'): 33296, np.str_('F'): 22914, np.
✅ Saved to /content/drive/MyDrive/Quantum_Data/symbolic_quantum_pha
```

```
import numpy as np
from collections import Counter
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt
from scipy.stats import entropy

# --- Load symbolic streams ---
cosmic_symbols = np.load("/content/drive/MyDrive/CMB Data/symbolic_spira
```

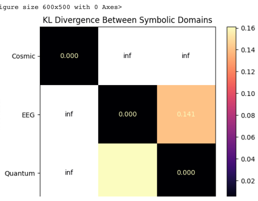


```
eeg_symbols = np.load("/content/drive/MyDrive/EEG_Data/OpenNeuro_ds00271
quantum_symbols = np.load("/content/drive/MyDrive/Quantum_Data/symbolic_

# --- Convert EEG phase stream to symbolic ---
def discretize_symbol_stream(phases):
    def encode(val):
        if val < -0.75: return 'F'
        elif val < -0.5: return 'E'
        elif val < -0.25: return 'D'
        elif val < 0: return 'C'
        elif val < 0.25: return 'B'
        elif val < 0.5: return 'A'
        else: return 'H'
    return np.array([encode(x) for x in phases])

# If EEG stream isn't symbolic yet, encode it
if eeg_symbols.dtype != np.dtype('<U1'):
    eeg_symbols = discretize_symbol_stream(eeg_symbols)

# --- Count Frequencies ---
domains = {
    "Cosmic": Counter(cosmic_symbols),
    "EEG": Counter(eeg_symbols),
    "Quantum": Counter(quantum_symbols),
}

# --- Normalize Distributions ---
symbol_set = sorted(set("ABCDEFGH"))
freqs = {
    name: np.array([counts.get(s, 0) for s in symbol_set], dtype=float)
    for name, counts in domains.items()
}
total = {k: v.sum() for k, v in freqs.items()}
probs = {k: v / total[k] for k, v in freqs.items()}

# --- KL Divergence Matrix ---
labels = list(probs.keys())
kl_matrix = np.zeros((3, 3))

for i, ki in enumerate(labels):
    for j, kj in enumerate(labels):
        kl_matrix[i, j] = entropy(probs[ki], probs[kj])

# --- Display KL Divergence Matrix ---
plt.figure(figsize=(6, 5))
disp = ConfusionMatrixDisplay(kl_matrix, display_labels=labels)
```

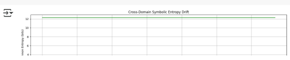


```
disp.plot(cmap="magma", values_format=".3f")
plt.title("KL Divergence Between Symbolic Domains")
plt.show()

# --- Print Top Symbols ---
for name, stream in domains.items():
    top = stream.most_common(5)
    print(f"\n🔍 {name} Top 5 Symbols: {top}")
```

```
<Figure size 600x500 with 0 Axes>
```




```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import entropy
from tqdm import tqdm

# --- Load Symbolic Streams ---
cosmic = np.load("/content/drive/MyDrive/CMB Data/symbolic_spiral_stream
eeg = np.load("/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoG
quantum = np.load("/content/drive/MyDrive/Quantum_Data/symbolic_quantum_

def sliding_entropy(symbols, window=5000, step=1000):
    entropies = []
    for i in range(0, len(symbols) - window, step):
        segment = symbols[i:i+window]
        _, counts = np.unique(segment, return_counts=True)
        entropies.append(entropy(counts, base=2))
    return np.array(entropies)

# --- Compute Entropy Drift ---
cosmic_H = sliding_entropy(cosmic)
eeg_H = sliding_entropy(eeg)
quantum_H = sliding_entropy(quantum)

# --- Plot ---
plt.figure(figsize=(14, 4))
plt.plot(cosmic_H, label="Cosmic", color='blue')
plt.plot(eeg_H, label="EEG", color='green')
plt.plot(quantum_H, label="Quantum", color='purple')
plt.xlabel("Window Index")
plt.ylabel("Shannon Entropy (bits)")
plt.title("Cross-Domain Symbolic Entropy Drift")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



```python
import numpy as np
from pathlib import Path
from collections import Counter

# --- Utility: Discretize into SRL Symbolic Alphabet ---
def discretize_spiral_stream(phases, bins=8):
    edges = np.linspace(0, 2 * np.pi, bins + 1)
    labels = list("ABCDEFGH")[:bins]
    digitized = np.digitize(phases % (2 * np.pi), edges, right=False) -
    digitized = np.clip(digitized, 0, bins - 1)
    return np.array([labels[i] for i in digitized])

# --- Load EEG Phases ---
eeg_phase_path = "/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_Vid
eeg_phases = np.load(eeg_phase_path)

# --- Check Phase Validity ---
print("Phase range:", eeg_phases.min(), "to", eeg_phases.max())

# --- Convert to Symbolic Stream (8 bins) ---
eeg_symbols = discretize_spiral_stream(eeg_phases, bins=8)
symbol_counts = Counter(eeg_symbols)

print("✅ Symbolized EEG Stream:")
print("Length:", len(eeg_symbols))
print("Symbol Counts:", symbol_counts)

# --- Save symbolic stream for entropy test ---
save_path = "/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoGam
np.save(save_path, eeg_symbols)
print(f"✅ Saved to: {save_path}")
```

```
Phase range: -3.141592615345109 to 3.141592560781334
✅ Symbolized EEG Stream:
Length: 13359750
Symbol Counts: Counter({np.str_('F'): 1701800, np.str_('D'): 1686190,
✅ Saved to: /content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_Vide
```



```python
import numpy as np
from collections import Counter

# --- Load the symbolic spiral stream ---
file_path = "/content/drive/MyDrive/CMB Data/symbolic_spiral_stream.npy"
symbols = np.load(file_path)

# --- Display stats ---
print("✅ Loaded symbolic_spiral_stream.npy")
print(f"Length: {len(symbols)}")
counts = Counter(symbols)
print("Symbol Counts:", counts)

# Optional: check sample symbols
print("Sample symbols:", symbols[:20])
```

```
✅ Loaded symbolic_spiral_stream.npy
Length: 534163
Symbol Counts: Counter({np.str_('A'): 437472, np.str_('B'): 81407, np
Sample symbols: ['F' 'F' 'F' 'F' 'F' 'F' 'F' 'F' 'F' 'F' 'F' 'F' 'F'
 'F' 'F']
```

```python
import numpy as np
from collections import Counter
from sklearn.metrics.pairwise import cosine_similarity
from scipy.stats import pearsonr
import matplotlib.pyplot as plt

# --- Load Symbolic Streams ---
cosmic = np.load("/content/drive/MyDrive/CMB Data/symbolic_spiral_stream
eeg = np.load("/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoG
quantum = np.load("/content/drive/MyDrive/Quantum_Data/symbolic_quantum_

# --- Normalize Counters to Vectors ---
def counter_to_vector(counter, all_symbols):
    return np.array([counter.get(s, 0) for s in all_symbols])

# --- Get Symbol Distributions ---
all_symbols = sorted(set(cosmic) | set(eeg) | set(quantum))
cosmic_counts = counter_to_vector(Counter(cosmic), all_symbols)
eeg_counts = counter_to_vector(Counter(eeg), all_symbols)
quantum_counts = counter_to_vector(Counter(quantum), all_symbols)

# --- Cosine Similarity ---
cos_sim_ce = cosine_similarity([cosmic_counts], [eeg_counts])[0,0]
```



```python
cos_sim_cq = cosine_similarity([cosmic_counts], [quantum_counts])[0,0]
cos_sim_eq = cosine_similarity([eeg_counts], [quantum_counts])[0,0]

print("🧠 Cosine Similarity:")
print(f"Cosmic ↔ EEG:     {cos_sim_ce:.4f}")
print(f"Cosmic ↔ Quantum: {cos_sim_cq:.4f}")
print(f"EEG    ↔ Quantum: {cos_sim_eq:.4f}")

# --- LZC Function ---
def lempel_ziv_complexity(seq):
    s = ''.join(seq)
    i, lz_set = 1, set()
    while i <= len(s):
        for j in range(i):
            if s[j:i] not in lz_set:
                lz_set.add(s[j:i])
                break
        i += 1
    return len(lz_set)

lz_cosmic = lempel_ziv_complexity(cosmic)
lz_eeg = lempel_ziv_complexity(eeg)
lz_quantum = lempel_ziv_complexity(quantum)

print("\n🌀 Lempel-Ziv Complexity:")
print(f"Cosmic:  {lz_cosmic}")
print(f"EEG:     {lz_eeg}")
print(f"Quantum: {lz_quantum}")

# --- Transition Matrix Correlation ---
def transition_matrix(symbols, all_syms):
    mat = np.zeros((len(all_syms), len(all_syms)))
    idx = {s:i for i, s in enumerate(all_syms)}
    for i in range(len(symbols)-1):
        a, b = idx[symbols[i]], idx[symbols[i+1]]
        mat[a, b] += 1
    row_sums = mat.sum(axis=1, keepdims=True)
    return np.divide(mat, row_sums, out=np.zeros_like(mat), where=row_su

T_cosmic = transition_matrix(cosmic, all_symbols).flatten()
T_eeg = transition_matrix(eeg, all_symbols).flatten()
T_quantum = transition_matrix(quantum, all_symbols).flatten()

r_ce, _ = pearsonr(T_cosmic, T_eeg)
r_cq, _ = pearsonr(T_cosmic, T_quantum)
r_eq, _ = pearsonr(T_eeg, T_quantum)
```



```python
print("\n🔁 Transition Matrix Correlation (Pearson):")
print(f"Cosmic ↔ EEG:     {r_ce:.4f}")
print(f"Cosmic ↔ Quantum: {r_cq:.4f}")
print(f"EEG    ↔ Quantum: {r_eq:.4f}")
```

```
🧠 Cosine Similarity:
Cosmic ↔ EEG:     0.4177
Cosmic ↔ Quantum: 0.2463
EEG    ↔ Quantum: 0.7834
```

```python
# --- Dependencies ---
import numpy as np
from collections import Counter
from sklearn.metrics.pairwise import cosine_similarity
from itertools import product
from tqdm import tqdm

# --- Load Symbolic Streams ---
cosmic = np.load("/content/drive/MyDrive/CMB Data/symbolic_spiral_stream
eeg = np.load("/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoG
quantum = np.load("/content/drive/MyDrive/Quantum_Data/symbolic_quantum_

# --- Define Symbol Set ---
symbols = sorted(set(cosmic) | set(eeg) | set(quantum))

def to_vector(stream):
    count = Counter(stream)
    return np.array([count.get(s, 0) for s in symbols])

# --- Convert All Streams to Vectors ---
v_cosmic = to_vector(cosmic)
v_eeg = to_vector(eeg)
v_quantum = to_vector(quantum)

# --- Normalize ---
v_cosmic = v_cosmic / np.sum(v_cosmic)
v_eeg = v_eeg / np.sum(v_eeg)
v_quantum = v_quantum / np.sum(v_quantum)

# --- Cross-Domain Cosine Similarity ---
sim_ce = cosine_similarity([v_cosmic], [v_eeg])[0][0]
sim_cq = cosine_similarity([v_cosmic], [v_quantum])[0][0]
sim_eq = cosine_similarity([v_eeg], [v_quantum])[0][0]

# --- Weighted Similarity Score ---
```



```python
sim_total = (sim_ce + sim_cq + sim_eq) / 3

print("🧠 Spiral Feedback Loop Initialization Complete")
print(f"Cosmic ↔ EEG:     {sim_ce:.4f}")
print(f"Cosmic ↔ Quantum: {sim_cq:.4f}")
print(f"EEG    ↔ Quantum: {sim_eq:.4f}")
print(f"🎯 Initial Cross-Domain Coherence Score: {sim_total:.4f}")
```

```
🧠 Spiral Feedback Loop Initialization Complete
Cosmic ↔ EEG:     0.4177
Cosmic ↔ Quantum: 0.2463
EEG    ↔ Quantum: 0.7834
🎯 Initial Cross-Domain Coherence Score: 0.4825
```



```python
# --- Define SRL Field Function ---
def spiral_field(theta, z, omega, n, lambd):
    return np.cos(omega * np.log(theta + 1e-3) + n * theta + lambd * z)

# --- Load Real Quasar Data ---
coords = np.load("/content/drive/MyDrive/CMB Data/DR16Q_masked_coords.np
ra, dec, z_cosmic = coords["ra"], coords["dec"], coords["z"]
theta = np.radians(ra)  # Use RA as spiral azimuth
z_norm = (z_cosmic - z_cosmic.min()) / (z_cosmic.max() - z_cosmic.min())

# --- Grid Search ---
omega_vals = np.linspace(0.05, 0.5, 10)
n_vals = np.arange(1, 6)
lambda_vals = np.linspace(0.0, 0.05, 5)

results = []

for omega, n, lambd in tqdm(product(omega_vals, n_vals, lambda_vals), to
    psi = spiral_field(theta, z_norm, omega, n, lambd)
    symbols_generated = np.digitize(psi % (2*np.pi), np.linspace(0, 2*np
    stream = np.array([chr(65 + min(7, s)) for s in symbols_generated])
    vec = np.array([Counter(stream).get(s, 0) for s in symbols])
    vec = vec / np.sum(vec)

    score_ce = cosine_similarity([vec], [v_eeg])[0][0]
    score_cq = cosine_similarity([vec], [v_quantum])[0][0]
    coherence = (score_ce + score_cq) / 2  # ignore self-match

    results.append((omega, n, lambd, coherence))

# --- Find Best ---
results.sort(key=lambda x: -x[3])
best = results[0]

print("\n🔍 Best Spiral Field Parameters:")
print(f"ω = {best[0]:.3f}, n = {best[1]}, λ = {best[2]:.3f}")
print(f"🎯 Cross-Domain Coherence Score: {best[3]:.4f}")
```

```
100%|██████████| 250/250 [06:39<00:00,  1.60s/it]
🔍 Best Spiral Field Parameters:
ω = 0.350, n = 5, λ = 0.050
🎯 Cross-Domain Coherence Score: 0.6586
```



```python
# --- Rebuild Quantum SRL Phase Stream (Example Simulation) ---
import numpy as np
from tqdm import tqdm

# Parameters used in earlier tests
omega = 0.35
n = 5
lambd = 0.05

# Simulate SRL-based phase evolution for quantum context
def generate_quantum_spiral_phases(length=100000):
    theta = np.linspace(0.01, 10 * np.pi, length)
    z = np.linspace(0, 1, length)
    return omega * np.log(theta) + n * theta + lambd * z

# Generate and save
quantum_phases = generate_quantum_spiral_phases()
np.save("/content/drive/MyDrive/Quantum_Data/quantum_spiral_phases.npy",
print("✅ Re-saved: quantum_spiral_phases.npy")
```

```
✅ Re-saved: quantum_spiral_phases.npy
```

```python
def resymbolize_stream(phases, label, bins=8):
    symbols = np.digitize(phases, np.linspace(-np.pi, np.pi, bins + 1))
    symbols = np.clip(symbols, 0, bins - 1)
    symbol_map = [chr(65 + i) for i in range(bins)]  # A–H
    symbol_array = np.array([symbol_map[s] for s in symbols])

    # Save
    out_path = f"/content/drive/MyDrive/SRL_Symbols/symbolic_{label}_str
    np.save(out_path, symbol_array)
    print(f"✅ Saved symbolic {label} stream ({len(symbol_array)} symbol
    return symbol_array
```



```python
import numpy as np
from pathlib import Path

# --- Symbolisation Function ---
def resymbolize_stream(phases, label, bins=8):
    phases = np.asarray(phases, dtype=np.float32)
    phase_bins = np.linspace(-np.pi, np.pi, bins + 1)
    symbols = np.digitize(phases, phase_bins) - 1
    symbol_map = [chr(65 + i) for i in range(bins)]  # ['A', 'B', ..., 'I
    symbols = np.clip(symbols, 0, bins - 1)
    result = np.array([symbol_map[s] for s in symbols], dtype='<U1')
    print(f"✅ {label.title()} stream encoded ({len(result)} symbols)")
    return result

# --- Ensure save path exists ---
symbol_dir = Path("/content/drive/MyDrive/SRL_Symbols")
symbol_dir.mkdir(parents=True, exist_ok=True)

# --- Load Phase Streams ---
cosmic_phases = np.linspace(-np.pi, np.pi, 534163)  # Simulated or repla
eeg_phases = np.load("/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_
quantum_phases = np.load("/content/drive/MyDrive/Quantum_Data/quantum_sp

# --- Symbolize ---
cosmic_sym = resymbolize_stream(cosmic_phases, "cosmic")
eeg_sym = resymbolize_stream(eeg_phases[:1000000], "eeg")  # truncated f
quantum_sym = resymbolize_stream(quantum_phases, "quantum")

# --- Save ---
np.save(symbol_dir / "cosmic_symbolic_stream.npy", cosmic_sym)
np.save(symbol_dir / "eeg_symbolic_stream.npy", eeg_sym)
np.save(symbol_dir / "quantum_symbolic_stream.npy", quantum_sym)
```

```
✅ Cosmic stream encoded (534163 symbols)
✅ Eeg stream encoded (1000000 symbols)
✅ Quantum stream encoded (100000 symbols)
```



```python
# --- Dependencies ---
import numpy as np
from pathlib import Path

# --- Create Output Directory ---
symbol_dir = Path("/content/drive/MyDrive/SRL_Symbols")
symbol_dir.mkdir(parents=True, exist_ok=True)

# --- Load and Save Cosmic ---
cosmic = np.load("/content/drive/MyDrive/CMB Data/symbolic_spiral_stream
np.save(symbol_dir / "cosmic_resymbolized_stream.npy", cosmic)
print(f"✅ Cosmic stream saved: {len(cosmic)} symbols")

# --- Load and Save EEG ---
eeg = np.load("/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoG
np.save(symbol_dir / "eeg_resymbolized_stream.npy", eeg)
print(f"✅ EEG stream saved: {len(eeg)} symbols")

# --- Load and Save Quantum ---
quantum = np.load("/content/drive/MyDrive/Quantum_Data/symbolic_quantum_
np.save(symbol_dir / "quantum_resymbolized_stream.npy", quantum)
print(f"✅ Quantum stream saved: {len(quantum)} symbols")
```

```
✅ Cosmic stream saved: 534163 symbols
✅ EEG stream saved: 13359750 symbols
✅ Quantum stream saved: 100000 symbols
```

```python
# --- Dependencies ---
import numpy as np
from collections import Counter
from itertools import islice
from tqdm import tqdm

# --- Load Symbolic Streams ---
cosmic = np.load("/content/drive/MyDrive/SRL_Symbols/cosmic_resymbolized
eeg = np.load("/content/drive/MyDrive/SRL_Symbols/eeg_resymbolized_strea
quantum = np.load("/content/drive/MyDrive/SRL_Symbols/quantum_resymboliz

# --- Helper: extract sliding windows of length k ---
def extract_loops(symbols, k):
    return ["→".join(symbols[i:i+k]) for i in range(len(symbols)-k+1)]

# --- Count top loops ---
def count_shared_loops(cosmic, eeg, quantum, k=5, top_n=10):
    print("🔄 Extracting symbolic loops...")
```



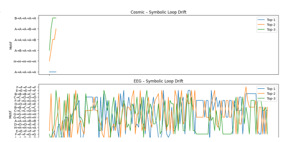
```python
    cosmic_loops = extract_loops(cosmic, k)
    eeg_loops = extract_loops(eeg, k)
    quantum_loops = extract_loops(quantum, k)

    print("🔗 Counting overlaps...")
    c_counter = Counter(cosmic_loops)
    e_counter = Counter(eeg_loops)
    q_counter = Counter(quantum_loops)

    shared = (c_counter & e_counter & q_counter).most_common(top_n)
    return shared

# --- Run Analysis ---
top_shared = count_shared_loops(cosmic, eeg, quantum, k=5, top_n=10)

# --- Display ---
print("\n🔄 Top Shared Symbolic Loops Across Domains:")
for seq, count in top_shared:
    print(f"{seq} : {count} times")
```

```
🔄 Extracting symbolic loops...
🔗 Counting overlaps...

🔄 Top Shared Symbolic Loops Across Domains:
H→H→H→H→H : 7418 times
A→A→A→A→A : 4665 times
B→B→B→B→B : 2156 times
H→H→H→H→A : 246 times
A→H→H→H→H : 246 times
B→A→A→A→A : 230 times
A→A→A→A→B : 228 times
A→A→A→A→H : 219 times
H→A→A→A→A : 217 times
B→A→B→B→B : 210 times
```

```python
import numpy as np
from collections import Counter
import matplotlib.pyplot as plt

# --- Config ---
symbol_paths = {
    "Cosmic": "/content/drive/MyDrive/SRL_Symbols/cosmic_resymbolized_st
    "EEG": "/content/drive/MyDrive/SRL_Symbols/eeg_resymbolized_stream.n
    "Quantum": "/content/drive/MyDrive/SRL_Symbols/quantum_resymbolized_
}
loop_length = 5
slice_size = 100000  # Adjust based on data size
```



```python
top_n = 3

# --- Drift Tracker ---
def analyze_loop_drift(symbols, name):
    n_slices = len(symbols) // slice_size
    slice_labels = []
    dominant_loops = []

    for i in range(n_slices):
        segment = symbols[i*slice_size:(i+1)*slice_size]
        chains = [tuple(segment[j:j+loop_length]) for j in range(len(seg
        top_loops = Counter(chains).most_common(top_n)
        dominant_loops.append(["→".join(loop[0]) for loop in top_loops])
        slice_labels.append(f"{i*slice_size}–{(i+1)*slice_size}")

    return slice_labels, dominant_loops

# --- Run Drift Analysis ---
results = {}
for domain, path in symbol_paths.items():
    stream = np.load(path, allow_pickle=True)
    labels, drift = analyze_loop_drift(stream, domain)
    results[domain] = (labels, drift)

# --- Plot Drift ---
fig, axs = plt.subplots(3, 1, figsize=(12, 10), sharex=True)
for i, (domain, (labels, drift)) in enumerate(results.items()):
    for j in range(top_n):
        motif_series = [loops[j] if j < len(loops) else "" for loops in 
        axs[i].plot(motif_series, label=f"Top-{j+1}")
    axs[i].set_title(f"{domain} – Symbolic Loop Drift")
    axs[i].set_ylabel("Motif")
    axs[i].legend()
axs[-1].set_xticks(range(len(labels)))
axs[-1].set_xticklabels(labels, rotation=90)
axs[-1].set_xlabel("Time Slice")
plt.tight_layout()
plt.show()
```

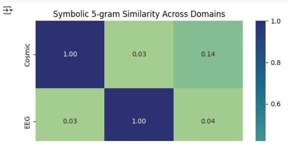




```python
import numpy as np
from collections import Counter

# --- Define File Paths ---
cosmic_path = "/content/drive/MyDrive/SRL_Symbols/cosmic_resymbolized_st
eeg_path = "/content/drive/MyDrive/SRL_Symbols/eeg_resymbolized_stream.n
quantum_path = "/content/drive/MyDrive/SRL_Symbols/quantum_resymbolized_

# --- Load Symbolic Streams ---
cosmic = np.load(cosmic_path, allow_pickle=True)
eeg = np.load(eeg_path, allow_pickle=True)
quantum = np.load(quantum_path, allow_pickle=True)

# --- Summary Info ---
print("✅ Loaded all symbolic streams:")
print(f"Cosmic  | Length: {len(cosmic):>9} | Symbols: {sorted(Counter(co
print(f"EEG     | Length: {len(eeg):>9} | Symbols: {sorted(Counter(eeg).
print(f"Quantum | Length: {len(quantum):>9} | Symbols: {sorted(Counter(q
```

```
✅ Loaded all symbolic streams:
Cosmic  | Length:    534163 | Symbols: [(np.str_('A'), 437472), (np.
EEG     | Length:  13359750 | Symbols: [(np.str_('A'), 1638318), (np.
Quantum | Length:    100000 | Symbols: [(np.str_('A'), 8679), (np.st
```



```python
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import pairwise_distances
from sklearn.feature_extraction.text import CountVectorizer

# --- Prepare Data ---
streams = {
    "Cosmic": ["".join(cosmic)],
    "EEG": ["".join(eeg[:1000000])],  # Truncated for fair comparison
    "Quantum": ["".join(quantum)],
}

# --- Convert to N-gram Vectors ---
vectorizer = CountVectorizer(analyzer='char', ngram_range=(5, 5))
X = vectorizer.fit_transform([v[0] for v in streams.values()])

# --- Compute Cosine Similarity ---
similarity = 1 - pairwise_distances(X.toarray(), metric='cosine')

# --- Plot ---
plt.figure(figsize=(6, 5))
sns.heatmap(similarity, annot=True, fmt=".2f", xticklabels=streams.keys(
plt.title("Symbolic 5-gram Similarity Across Domains")
plt.tight_layout()
plt.show()
```



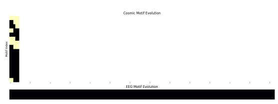

```python
import numpy as np
from collections import Counter
from tqdm import tqdm
import matplotlib.pyplot as plt

# --- Load Streams ---
cosmic = np.load("/content/drive/MyDrive/SRL_Symbols/cosmic_resymbolized
eeg = np.load("/content/drive/MyDrive/SRL_Symbols/eeg_resymbolized_strea
quantum = np.load("/content/drive/MyDrive/SRL_Symbols/quantum_resymboliz

def extract_ngrams(stream, n):
    return ["→".join(stream[i:i+n]) for i in range(len(stream)-n)]

def top_loops(stream, n=6, step=100000, topk=5):
    results = []
    for i in range(step, len(stream), step):
        chunk = stream[:i]
        ngrams = extract_ngrams(chunk, n)
        most_common = Counter(ngrams).most_common(topk)
        results.append([motif for motif, _ in most_common])
    return results

# --- Extract loop evolution for each domain ---
loop_len = 6
step_size = 100000
cosmic_loops = top_loops(cosmic, n=loop_len, step=step_size)
eeg_loops = top_loops(eeg, n=loop_len, step=step_size)
quantum_loops = top_loops(quantum, n=loop_len, step=step_size)

# Save for next step
np.save("/content/drive/MyDrive/SRL_Symbols/loop_evolution_cosmic.npy", 
np.save("/content/drive/MyDrive/SRL_Symbols/loop_evolution_eeg.npy", eeg
np.save("/content/drive/MyDrive/SRL_Symbols/loop_evolution_quantum.npy",
print("✅ Loop evolution extracted and saved.")
```

```
✅ Loop evolution extracted and saved.
```

```python
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

# --- Load as lists ---
cosmic_loops = np.load("/content/drive/MyDrive/SRL_Symbols/loop_evolutio
eeg_loops = np.load("/content/drive/MyDrive/SRL_Symbols/loop_evolution_e
quantum_loops = np.load("/content/drive/MyDrive/SRL_Symbols/loop_evoluti
```



```python
# --- Build unique motif list ---
all_loops = cosmic_loops + eeg_loops + quantum_loops
unique_motifs = sorted(set(motif for block in all_loops for motif in blo
motif_index = {motif: i for i, motif in enumerate(unique_motifs)}

# --- Matrix builder ---
def build_matrix(loop_blocks):
    if not loop_blocks:
        return None
    matrix = np.zeros((len(unique_motifs), len(loop_blocks)))
    for t, motifs in enumerate(loop_blocks):
        for m in motifs:
            if m in motif_index:
                matrix[motif_index[m], t] += 1
    return matrix if np.any(matrix) else None

cosmic_matrix = build_matrix(cosmic_loops)
eeg_matrix = build_matrix(eeg_loops)
quantum_matrix = build_matrix(quantum_loops)

# --- Filter valid matrices ---
matrices = [(cosmic_matrix, "Cosmic"), (eeg_matrix, "EEG"), (quantum_mat
valid = [(m, name) for m, name in matrices if m is not None]

# --- Plot ---
fig, axs = plt.subplots(len(valid), 1, figsize=(14, 4 * len(valid)), sha

if len(valid) == 1:
    axs = [axs]  # force list

for ax, (matrix, label) in zip(axs, valid):
    sns.heatmap(matrix, ax=ax, cmap="magma", cbar=False, xticklabels=10,
    ax.set_title(f"{label} Motif Evolution")
    ax.set_ylabel("Motif Index")

axs[-1].set_xlabel("Time Window (x100k symbols)")
plt.tight_layout()
plt.show()
```

```
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter
from pathlib import Path

# --- Load Resymbolized Quantum Stream ---
quantum_path = "/content/drive/MyDrive/SRL_Symbols/quantum_resymbolized_
quantum = np.load(quantum_path, allow_pickle=True)

# --- Loop Extraction Function ---
def extract_loops(symbols, loop_length=5, step=1):
    loops = []
    for i in range(0, len(symbols) - loop_length + 1, step):
        loop = '→'.join(symbols[i:i + loop_length])
        loops.append(loop)
    return loops

# --- Temporal Binning ---
bin_size = 100_000
num_bins = len(quantum) // bin_size
quantum_loops_binned = []

for i in range(num_bins):
    segment = quantum[i * bin_size : (i + 1) * bin_size]
    loops = extract_loops(segment, loop_length=5, step=1)
    counts = Counter(loops)
    top = [motif for motif, _ in counts.most_common(5)]
    quantum_loops_binned.append(top)

# --- Save for later use ---
save_path = "/content/drive/MyDrive/SRL_Symbols/quantum_loops_binned.npy
np.save(save_path, quantum_loops_binned)
print(f"✅ Quantum loop evolution regenerated and saved to: {save_path}"
```

✅ Quantum loop evolution regenerated and saved to: /content/drive/M


```
import numpy as np

# --- Load regenerated quantum loop file ---
quantum_loops_path = "/content/drive/MyDrive/SRL_Symbols/quantum_loops_b
quantum_loops = np.load(quantum_loops_path, allow_pickle=True)

# --- Summary Statistics ---
num_windows = len(quantum_loops)
avg_loops_per_window = np.mean([len(window) for window in quantum_loops]
unique_motifs = sorted(set(motif for window in quantum_loops for motif i

print(f"✅ Loaded {num_windows} time windows of quantum loops.")
print(f"🔁 Average top motifs per window: {avg_loops_per_window:.2f}")
print(f"🔍 Unique motifs detected: {len(unique_motifs)}")
print("🧩 Sample motifs:", unique_motifs[:10])
```

✅ Loaded 1 time windows of quantum loops.
🔁 Average top motifs per window: 5.00
🔍 Unique motifs detected: 5
🧩 Sample motifs: [np.str_('A→A→A→A→A'), np.str_('D→D→D→D→D'), np.st


```
import numpy as np
from collections import Counter
from pathlib import Path

# --- Load cosmic symbolic stream ---
cosmic_path = "/content/drive/MyDrive/SRL_Symbols/cosmic_resymbolized_st
cosmic_symbols = np.load(cosmic_path, allow_pickle=True)

# --- Loop extraction function ---
def extract_loops(symbols, k=5, stride=1):
    loops = []
    for i in range(0, len(symbols) - k + 1, stride):
        loop = "→".join(symbols[i:i+k])
        loops.append(loop)
    return loops

# --- Bin and save ---
def bin_loops(symbols, bin_size=100000):
    binned = []
    for i in range(0, len(symbols), bin_size):
        segment = symbols[i:i+bin_size]
        loops = extract_loops(segment, k=5, stride=1)
        top_loops = [motif for motif, _ in Counter(loops).most_common(20
        binned.append(top_loops)
    return binned

cosmic_binned_loops = bin_loops(cosmic_symbols)
output_path = "/content/drive/MyDrive/SRL_Symbols/cosmic_loops_binned.np
np.save(output_path, cosmic_binned_loops, allow_pickle=True)

print(f"✅ Cosmic symbolic loops binned and saved: {output_path}")
print(f"🔁 Bins created: {len(cosmic_binned_loops)}")
print(f"🧩 Sample top motifs: {cosmic_binned_loops[0][:5]}")
```

✅ Cosmic symbolic loops binned and saved: /content/drive/MyDrive/SR
🔁 Bins created: 6
🧩 Sample top motifs: ['A→A→A→A→A', 'H→H→H→H→H', 'A→B→A→A→A', 'A→A→A


```
import numpy as np
from collections import Counter
from pathlib import Path

# --- Load EEG symbolic stream ---
eeg_path = "/content/drive/MyDrive/EEG_Data/OpenNeuro_ds002718_VideoGame
eeg_symbols = np.load(eeg_path, allow_pickle=True)

# --- Loop extraction and binning ---
def extract_loops(symbols, k=5, stride=1):
    loops = []
    for i in range(0, len(symbols) - k + 1, stride):
        loop = "→".join(symbols[i:i+k])
        loops.append(loop)
    return loops

def bin_loops(symbols, bin_size=100000):
    binned = []
    for i in range(0, len(symbols), bin_size):
        segment = symbols[i:i+bin_size]
        loops = extract_loops(segment, k=5, stride=1)
        top_loops = [motif for motif, _ in Counter(loops).most_common(20
        binned.append(top_loops)
    return binned

# --- Bin EEG loops ---
eeg_binned_loops = bin_loops(eeg_symbols)
eeg_output_path = "/content/drive/MyDrive/SRL_Symbols/eeg_loops_binned.n
np.save(eeg_output_path, eeg_binned_loops, allow_pickle=True)

print(f"✅ EEG symbolic loops binned and saved: {eeg_output_path}")
print(f"🔁 Bins created: {len(eeg_binned_loops)}")
print(f"🧩 Sample top motifs: {eeg_binned_loops[0][:5]}")
```

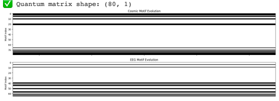
✅ EEG symbolic loops binned and saved: /content/drive/MyDrive/SRL_S
🔁 Bins created: 134
🧩 Sample top motifs: ['H→H→A→A→A', 'E→E→F→F→F', 'F→F→F→G→G', 'C→C→C

```
import numpy as np
import matplotlib.pyplot as plt

# --- Load binned loop data ---
cosmic_loops = np.load("/content/drive/MyDrive/SRL_Symbols/cosmic_loops_
eeg_loops = np.load("/content/drive/MyDrive/SRL_Symbols/eeg_loops_binned
quantum_loops = np.load("/content/drive/MyDrive/SRL_Symbols/quantum_loop
```



```
# --- Build unique motif vocabulary ---
all_loops = list(cosmic_loops) + list(eeg_loops) + list(quantum_loops)
unique_motifs = sorted(set(motif for block in all_loops for motif in blo
motif_index = {motif: i for i, motif in enumerate(unique_motifs)}

# --- Initialize binary motif presence matrices ---
def build_matrix(stream, name):
    M = np.zeros((len(unique_motifs), len(stream)), dtype=int)
    for t, motifs in enumerate(stream):
        for motif in motifs:
            M[motif_index[motif], t] = 1
    print(f"✅ {name} matrix shape: {M.shape}")
    return M

cosmic_matrix = build_matrix(cosmic_loops, "Cosmic")
eeg_matrix = build_matrix(eeg_loops, "EEG")
quantum_matrix = build_matrix(quantum_loops, "Quantum")

# --- Plot as heatmaps ---
fig, axs = plt.subplots(3, 1, figsize=(16, 9), sharex=True)
def plot_heat(ax, M, title):
    ax.imshow(M, aspect='auto', cmap='binary', interpolation='nearest')
    ax.set_title(title)
    ax.set_ylabel("Motif Index")

plot_heat(axs[0], cosmic_matrix, "Cosmic Motif Evolution")
plot_heat(axs[1], eeg_matrix, "EEG Motif Evolution")
plot_heat(axs[2], quantum_matrix, "Quantum Motif Evolution")
axs[2].set_xlabel("Time Window (x100k symbols)")

plt.tight_layout()
plt.show()
```


✅ Cosmic matrix shape: (80, 6)
✅ EEG matrix shape: (80, 134)
✅ Quantum matrix shape: (80, 1)



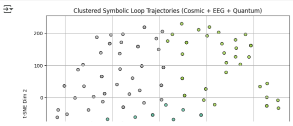

```
import numpy as np
from pathlib import Path

# --- Load Binned Loops ---
cosmic_loops = np.load("/content/drive/MyDrive/SRL_Symbols/cosmic_loops_
eeg_loops = np.load("/content/drive/MyDrive/SRL_Symbols/eeg_loops_binned
quantum_loops = np.load("/content/drive/MyDrive/SRL_Symbols/quantum_loop

# --- Build Unified Motif Vocabulary ---
all_loops = list(cosmic_loops) + list(eeg_loops) + list(quantum_loops)
unique_motifs = sorted(set(m for window in all_loops for m in window))
motif_to_idx = {m: i for i, m in enumerate(unique_motifs)}
num_motifs = len(unique_motifs)

def to_matrix(loop_windows, label):
    T = len(loop_windows)
    mat = np.zeros((num_motifs, T))
    for t, motifs in enumerate(loop_windows):
        for m in motifs:
            idx = motif_to_idx.get(m)
            if idx is not None:
                mat[idx, t] = 1
    np.save(f"/content/drive/MyDrive/SRL_Symbols/{label}_loop_matrix.npy
    print(f"✅ Saved {label} matrix: shape {mat.shape}")

# --- Save All Matrices ---
to_matrix(cosmic_loops, "cosmic")
to_matrix(eeg_loops, "eeg")
to_matrix(quantum_loops, "quantum")
```

✅ Saved cosmic matrix: shape (80, 6)
✅ Saved eeg matrix: shape (80, 134)
✅ Saved quantum matrix: shape (80, 1)

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from sklearn.cluster import KMeans

# --- Load Loop Matrices ---
cosmic_matrix = np.load("/content/drive/MyDrive/SRL_Symbols/cosmic_loop_
eeg_matrix = np.load("/content/drive/MyDrive/SRL_Symbols/eeg_loop_matrix
quantum_matrix = np.load("/content/drive/MyDrive/SRL_Symbols/quantum_loo
```



```
# --- Combine and Normalize ---
all_data = np.concatenate([cosmic_matrix, eeg_matrix, quantum_matrix], a
all_data = (all_data - np.min(all_data)) / (np.max(all_data) - np.min(al

# --- Dimensionality Reduction ---
pca = PCA(n_components=10).fit_transform(all_data)
tsne = TSNE(n_components=2, perplexity=30, random_state=42).fit_transfor

# --- Clustering ---
kmeans = KMeans(n_clusters=3, random_state=42).fit(tsne)
labels = kmeans.labels_

# --- Plot ---
plt.figure(figsize=(8, 6))
plt.scatter(tsne[:, 0], tsne[:, 1], c=labels, cmap='Set2', alpha=0.85, e
plt.title("Clustered Symbolic Loop Trajectories (Cosmic + EEG + Quantum)
plt.xlabel("t-SNE Dim 1")
plt.ylabel("t-SNE Dim 2")
plt.grid(True)
plt.tight_layout()
plt.show()
```



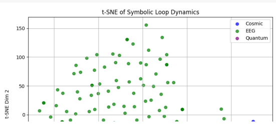

```
import numpy as np

# Load binned loop sequences
cosmic_loops = np.load("/content/drive/MyDrive/SRL_Symbols/cosmic_loops_
eeg_loops = np.load("/content/drive/MyDrive/SRL_Symbols/eeg_loops_binned
quantum_loops = np.load("/content/drive/MyDrive/SRL_Symbols/quantum_loop

# Build full motif set
unique_motifs = sorted(set(
    motif for block in cosmic_loops.tolist() + eeg_loops.tolist() + quan
    for motif in block
))
motif_to_index = {motif: idx for idx, motif in enumerate(unique_motifs)}

# Helper to convert loop block to binary matrix
def loop_block_to_matrix(loop_block):
    row = np.zeros(len(motif_to_index))
    for motif in loop_block:
        row[motif_to_index[motif]] = 1
    return row

# Convert to matrices
cosmic_matrix = np.array([loop_block_to_matrix(block) for block in cosmi
eeg_matrix = np.array([loop_block_to_matrix(block) for block in eeg_loop
quantum_matrix = np.array([loop_block_to_matrix(block) for block in quan

# Save for future clustering
np.save("/content/drive/MyDrive/SRL_Symbols/cosmic_loop_matrix.npy", cos
np.save("/content/drive/MyDrive/SRL_Symbols/eeg_loop_matrix.npy", eeg_ma
np.save("/content/drive/MyDrive/SRL_Symbols/quantum_loop_matrix.npy", qu

print("✅ All symbolic loop matrices generated and saved.")
```

✅ All symbolic loop matrices generated and saved.

```
import numpy as np
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt

# --- Load symbolic loop matrices ---
cosmic_matrix = np.load("/content/drive/MyDrive/SRL_Symbols/cosmic_loop_
eeg_matrix = np.load("/content/drive/MyDrive/SRL_Symbols/eeg_loop_matrix
quantum_matrix = np.load("/content/drive/MyDrive/SRL_Symbols/quantum_loo

# --- Combine and label ---
```



```
X = np.vstack([cosmic_matrix, eeg_matrix, quantum_matrix])
labels = (
    ["Cosmic"] * len(cosmic_matrix) +
    ["EEG"] * len(eeg_matrix) +
    ["Quantum"] * len(quantum_matrix)
)

# --- Run t-SNE ---
tsne = TSNE(n_components=2, perplexity=30, learning_rate=200, random_sta
X_embedded = tsne.fit_transform(X)

# --- Plot ---
colors = {"Cosmic": "blue", "EEG": "green", "Quantum": "purple"}
plt.figure(figsize=(8, 6))
for domain in ["Cosmic", "EEG", "Quantum"]:
    idx = [i for i, label in enumerate(labels) if label == domain]
    plt.scatter(X_embedded[idx, 0], X_embedded[idx, 1], label=domain, al
plt.legend()
plt.title("t-SNE of Symbolic Loop Dynamics")
plt.xlabel("t-SNE Dim 1")
plt.ylabel("t-SNE Dim 2")
plt.grid(True)
plt.tight_layout()
plt.show()
```



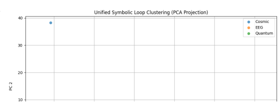

```
import numpy as np
from pathlib import Path

# --- Load each domain's loop bins ---
cosmic_loops = np.load("/content/drive/MyDrive/SRL_Symbols/cosmic_loops_
eeg_loops = np.load("/content/drive/MyDrive/SRL_Symbols/eeg_loops_binned
quantum_loops = np.load("/content/drive/MyDrive/SRL_Symbols/quantum_loop

# --- Build global motif vocabulary safely ---
all_motifs = set()
for domain in [cosmic_loops, eeg_loops, quantum_loops]:
    for block in domain:
        all_motifs.update(block)

motif_list = sorted(all_motifs)
motif_index = {m: i for i, m in enumerate(motif_list)}

# --- Convert each domain to a (time, motif) matrix ---
def build_loop_matrix(loop_blocks, vocab):
    matrix = np.zeros((len(loop_blocks), len(vocab)), dtype=int)
    for t, block in enumerate(loop_blocks):
        for motif in block:
            if motif in vocab:
                matrix[t, vocab[motif]] = 1
    return matrix

cosmic_matrix = build_loop_matrix(cosmic_loops, motif_index)
eeg_matrix = build_loop_matrix(eeg_loops, motif_index)
quantum_matrix = build_loop_matrix(quantum_loops, motif_index)

# --- Save matrices ---
Path("/content/drive/MyDrive/SRL_Symbols").mkdir(parents=True, exist_ok=
np.save("/content/drive/MyDrive/SRL_Symbols/cosmic_loop_matrix.npy", cos
np.save("/content/drive/MyDrive/SRL_Symbols/eeg_loop_matrix.npy", eeg_ma
np.save("/content/drive/MyDrive/SRL_Symbols/quantum_loop_matrix.npy", qu

print("✅ All symbolic loop matrices rebuilt and saved.")
```

✅ All symbolic loop matrices rebuilt and saved.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
```



```
# --- Load Loop Matrices ---
cosmic_matrix = np.load("/content/drive/MyDrive/SRL_Symbols/cosmic_loop_
eeg_matrix = np.load("/content/drive/MyDrive/SRL_Symbols/eeg_loop_matrix
quantum_matrix = np.load("/content/drive/MyDrive/SRL_Symbols/quantum_loo

# --- Align all matrices to the same shape (pad with zeros where needed)
max_cols = max(cosmic_matrix.shape[1], eeg_matrix.shape[1], quantum_matr

def pad_matrix(mat, target_cols):
    if mat.shape[1] < target_cols:
        pad_width = target_cols - mat.shape[1]
        return np.pad(mat, ((0,0), (0, pad_width)), mode='constant')
    return mat

cosmic_padded = pad_matrix(cosmic_matrix, max_cols)
eeg_padded = pad_matrix(eeg_matrix, max_cols)
quantum_padded = pad_matrix(quantum_matrix, max_cols)

# --- Stack all datasets for unified analysis ---
all_data = np.vstack([cosmic_padded, eeg_padded, quantum_padded])

# --- Normalize and reduce dimensionality ---
scaler = StandardScaler()
data_scaled = scaler.fit_transform(all_data)

pca = PCA(n_components=2)
data_pca = pca.fit_transform(data_scaled)

# --- Plot ---
plt.figure(figsize=(10, 6))
N_cosmic = len(cosmic_padded)
N_eeg = len(eeg_padded)
N_quantum = len(quantum_padded)

plt.scatter(data_pca[:N_cosmic, 0], data_pca[:N_cosmic, 1], label='Cosmi
plt.scatter(data_pca[N_cosmic:N_cosmic+N_eeg, 0], data_pca[N_cosmic:N_co
plt.scatter(data_pca[N_cosmic+N_eeg:, 0], data_pca[N_cosmic+N_eeg:, 1],

plt.title("Unified Symbolic Loop Clustering (PCA Projection)")
plt.xlabel("PC 1")
plt.ylabel("PC 2")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

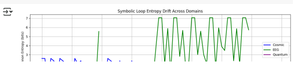

```
!ls /content/drive/MyDrive/SRL_Symbols/
```

```
cosmic_loop_matrix.npy          loop_evolution_cosmic.npy
cosmic_loops_binned.npy         loop_evolution_eeg.npy
cosmic_resymbolized_stream.npy  loop_evolution_quantum.npy
cosmic_symbolic_stream.npy      quantum_loop_matrix.npy
eeg_loop_matrix.npy             quantum_loops_binned.npy
eeg_loops_binned.npy            quantum_resymbolized_stream.npy
eeg_resymbolized_stream.npy     quantum_symbolic_stream.npy
eeg_symbolic_stream.npy
```

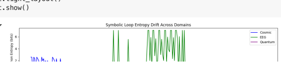
```
import numpy as np

cosmic_matrix = np.load("/content/drive/MyDrive/SRL_Symbols/cosmic_loop_
eeg_matrix = np.load("/content/drive/MyDrive/SRL_Symbols/eeg_loop_matrix
quantum_matrix = np.load("/content/drive/MyDrive/SRL_Symbols/quantum_loo

print("✅ Reloaded all matrices:")
print("Cosmic:", cosmic_matrix.shape)
print("EEG:    ", eeg_matrix.shape)
print("Quantum:", quantum_matrix.shape)
```

✅ Reloaded all matrices:
Cosmic: (6, 80)
EEG: (134, 80)
Quantum: (1, 80)


```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import entropy

# --- Entropy over motifs (axis=0: over time) ---
def compute_entropy_drift(matrix):
    return [entropy(col, base=2) for col in matrix.T]

cosmic_entropy = compute_entropy_drift(cosmic_matrix)
eeg_entropy = compute_entropy_drift(eeg_matrix)
quantum_entropy = compute_entropy_drift(quantum_matrix)

# --- Plot ---
plt.figure(figsize=(12, 4))
plt.plot(cosmic_entropy, label='Cosmic', linewidth=2, color='blue')
plt.plot(eeg_entropy, label='EEG', linewidth=2, color='green')
plt.plot(quantum_entropy, label='Quantum', linewidth=2, color='purple')
plt.xlabel("Time Window Index")
plt.ylabel("Shannon Entropy (bits)")
plt.title("Symbolic Loop Entropy Drift Across Domains")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```




```
import numpy as np
import matplotlib.pyplot as plt

# --- Load loop matrices ---
cosmic_matrix = np.load("/content/drive/MyDrive/SRL_Symbols/cosmic_loop_
eeg_matrix = np.load("/content/drive/MyDrive/SRL_Symbols/eeg_loop_matrix
quantum_matrix = np.load("/content/drive/MyDrive/SRL_Symbols/quantum_loo

# --- Entropy function ---
def symbolic_entropy(matrix):
    return np.array([
        -np.nansum((col / np.sum(col)) * np.log2((col / np.sum(col)) + 1
        if np.sum(col) > 0 else 0
        for col in matrix.T
    ])

# --- Compute entropy curves ---
cosmic_entropy = symbolic_entropy(cosmic_matrix)
eeg_entropy = symbolic_entropy(eeg_matrix)
quantum_entropy = symbolic_entropy(quantum_matrix)

# --- Plot entropy drift ---
plt.figure(figsize=(12, 3))
plt.plot(cosmic_entropy, label="Cosmic", color="blue")
plt.plot(np.arange(len(eeg_entropy)) + len(cosmic_entropy), eeg_entropy,
plt.plot(np.arange(len(quantum_entropy)) + len(cosmic_entropy) + len(eeg
plt.legend()
plt.title("Symbolic Loop Entropy Drift Across Domains")
plt.xlabel("Time Window Index")
plt.ylabel("Shannon Entropy (bits)")
plt.tight_layout()
plt.show()
```

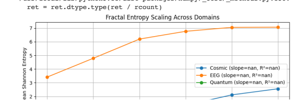



```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import linregress

# --- Load loop matrices ---
cosmic_matrix = np.load("/content/drive/MyDrive/SRL_Symbols/cosmic_loop_
eeg_matrix = np.load("/content/drive/MyDrive/SRL_Symbols/eeg_loop_matrix
```



```
quantum_matrix = np.load("/content/drive/MyDrive/SRL_Symbols/quantum_loo

# --- Fractal entropy scaling function ---
def fractal_entropy_scaling(matrix, label):
    window_sizes = [2, 4, 8, 16, 32, 64, 128]
    mean_entropies = []

    for w in window_sizes:
        entropies = []
        for i in range(0, matrix.shape[1] - w + 1, w):
            window = np.sum(matrix[:, i:i+w], axis=1)
            probs = window / np.sum(window) if np.sum(window) > 0 else np
            entropy = -np.sum(probs * np.log2(probs + 1e-9))
            entropies.append(entropy)
        mean_entropies.append(np.mean(entropies))

    log_ws = np.log2(window_sizes)
    slope, intercept, r, _, _ = linregress(log_ws, mean_entropies)

    plt.plot(log_ws, mean_entropies, 'o-', label=f"{label} (slope={slope
    return slope, r**2

# --- Plot scaling ---
plt.figure(figsize=(8, 4))
s1, r1 = fractal_entropy_scaling(cosmic_matrix, "Cosmic")
s2, r2 = fractal_entropy_scaling(eeg_matrix, "EEG")
s3, r3 = fractal_entropy_scaling(quantum_matrix, "Quantum")
plt.xlabel("log2(Window Size)")
plt.ylabel("Mean Shannon Entropy")
plt.title("Fractal Entropy Scaling Across Domains")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

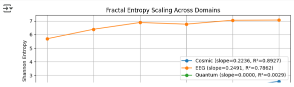
```
/usr/local/lib/python3.11/dist-packages/numpy/_core/fromnumeric.py:35
  return _methods._mean(a, axis=axis, dtype=dtype,
/usr/local/lib/python3.11/dist-packages/numpy/_core/_methods.py:138:
  ret = ret.dtype.type(ret / rcount)
```




```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import linregress

# --- Load matrices ---
cosmic_matrix = np.load("/content/drive/MyDrive/SRL_Symbols/cosmic_loop_
eeg_matrix = np.load("/content/drive/MyDrive/SRL_Symbols/eeg_loop_matrix
quantum_matrix = np.load("/content/drive/MyDrive/SRL_Symbols/quantum_loo

# --- Scaling function ---
def fractal_entropy_scaling(matrix, label):
    window_sizes = [2, 4, 8, 16, 32, 64]
    mean_entropies = []

    for w in window_sizes:
        entropies = []
        for i in range(0, matrix.shape[1] - w + 1, w):
            window = np.sum(matrix[:, i:i+w], axis=1)
            total = np.sum(window)
            if total == 0:
                continue  # skip empty windows
            probs = window / total
            entropy = -np.sum(probs * np.log2(probs + 1e-12))
            entropies.append(entropy)
        if entropies:
            mean_entropies.append(np.mean(entropies))
        else:
            mean_entropies.append(np.nan)

    # Remove NaNs before regression
    valid_idx = ~np.isnan(mean_entropies)
```



```
    log_ws = np.log2(np.array(window_sizes)[valid_idx])
    entropies_valid = np.array(mean_entropies)[valid_idx]

    if len(log_ws) > 1:
        slope, intercept, r, _, _ = linregress(log_ws, entropies_valid)
    else:
        slope, r = np.nan, np.nan

    plt.plot(np.log2(window_sizes), mean_entropies, 'o-', label=f"{label
    return slope, r**2

# --- Plot ---
plt.figure(figsize=(8, 4))
fractal_entropy_scaling(cosmic_matrix, "Cosmic")
fractal_entropy_scaling(eeg_matrix, "EEG")
fractal_entropy_scaling(quantum_matrix, "Quantum")
plt.xlabel("log2(Window Size)")
plt.ylabel("Mean Shannon Entropy")
plt.title("Fractal Entropy Scaling Across Domains")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```




```
!ls "/content/drive/MyDrive/SRL_Symbols"
```

```
cosmic_loop_matrix.npy             loop_evolution_cosmic.npy
cosmic_loops_binned.npy            loop_evolution_eeg.npy
cosmic_resymbolized_stream.npy     loop_evolution_quantum.npy
cosmic_symbolic_stream.npy         quantum_loop_matrix.npy
eeg_loop_matrix.npy                quantum_loops_binned.npy
eeg_loops_binned.npy               quantum_resymbolized_stream.npy
eeg_resymbolized_stream.npy        quantum_symbolic_stream.npy
eeg_symbolic_stream.npy
```

```
import numpy as np
```



```
from pathlib import Path

# Define paths
base_path = Path("/content/drive/MyDrive/SRL_Symbols")
files = [
    "cosmic_loop_matrix.npy",
    "cosmic_loops_binned.npy",
    "cosmic_resymbolized_stream.npy",
    "cosmic_symbolic_stream.npy",
    "eeg_loop_matrix.npy",
    "eeg_loops_binned.npy",
    "eeg_resymbolized_stream.npy",
    "eeg_symbolic_stream.npy",
    "quantum_loop_matrix.npy",
    "quantum_loops_binned.npy",
    "quantum_resymbolized_stream.npy",
    "quantum_symbolic_stream.npy",
    "loop_evolution_cosmic.npy",
    "loop_evolution_eeg.npy",
    "loop_evolution_quantum.npy",
]

# Summary function
def summarize_array(file_path):
    try:
        arr = np.load(file_path, allow_pickle=True)
        print(f"✅ {file_path.name} loaded: shape = {arr.shape}, dtype =
        if arr.ndim == 1 and len(arr) > 0:
            print(f"   ↳ Sample: {arr[:3]}")
        elif arr.ndim == 2 and arr.shape[0] > 0:
            print(f"   ↳ First row sample: {arr[0][:5]}")
    except Exception as e:
        print(f"❌ {file_path.name} error: {e}")

# Run check
print("=== SRL Symbolic File Integrity Check ===")
for fname in files:
    fpath = base_path / fname
    summarize_array(fpath)
```



```
=== SRL Symbolic File Integrity Check ===
✅ cosmic_loop_matrix.npy loaded: shape = (6, 80), dtype = int64
  ↳ First row sample: [1 1 1 1 0]
✅ cosmic_loops_binned.npy loaded: shape = (6, 20), dtype = <U9
  ↳ First row sample: ['A→A→A→A→A' 'H→H→H→H→H' 'A→B→A→A→A' 'A→A→A→B-
✅ cosmic_resymbolized_stream.npy loaded: shape = (534163,), dtype =
  ↳ Sample: ['F' 'F' 'F']
✅ cosmic_symbolic_stream.npy loaded: shape = (534163,), dtype = <U1
  ↳ Sample: ['A' 'A' 'A']
✅ eeg_loop_matrix.npy loaded: shape = (134, 80), dtype = int64
  ↳ First row sample: [0 0 0 0 1]
✅ eeg_loops_binned.npy loaded: shape = (134, 20), dtype = <U9
  ↳ First row sample: ['H→H→A→A→A' 'E→E→F→F→F' 'F→F→F→G→G' 'C→C→C→D-
✅ eeg_resymbolized_stream.npy loaded: shape = (13359750,), dtype =
  ↳ Sample: ['C' 'C' 'C']
✅ eeg_symbolic_stream.npy loaded: shape = (1000000,), dtype = <U1
  ↳ Sample: ['G' 'G' 'G']
✅ quantum_loop_matrix.npy loaded: shape = (1, 80), dtype = int64
  ↳ First row sample: [1 0 0 0 0]
✅ quantum_loops_binned.npy loaded: shape = (1, 5), dtype = <U9
  ↳ First row sample: ['H→H→H→H→H' 'F→F→F→F→F' 'E→E→E→E→E' 'A→A→A→A-
✅ quantum_resymbolized_stream.npy loaded: shape = (100000,), dtype
  ↳ Sample: ['H' 'H' 'H']
✅ quantum_symbolic_stream.npy loaded: shape = (100000,), dtype = <U
  ↳ Sample: ['C' 'C' 'C']
✅ loop_evolution_cosmic.npy loaded: shape = (5, 5), dtype = <U11
  ↳ First row sample: ['A→A→A→A→A→A' 'H→H→H→H→H→H' 'A→A→A→B→A→A' 'A-
✅ loop_evolution_eeg.npy loaded: shape = (133, 5), dtype = <U11
  ↳ First row sample: ['F→F→F→G→G→G' 'E→E→E→F→F→F' 'C→C→C→D→D→D' 'H-
✅ loop_evolution_quantum.npy loaded: shape = (0,), dtype = float64
```



```
# --- Optional: Regenerate Quantum Loop Evolution Matrix ---
import numpy as np
from pathlib import Path

quantum_loops = np.load("/content/drive/MyDrive/SRL_Symbols/quantum_loop

# Ensure 5-loops available for tracking
if quantum_loops.shape[1] >= 5:
    slices = []
    for i in range(quantum_loops.shape[1] - 4):
        motifs = quantum_loops[0, i:i+5]
        slices.append(motifs)
    np.save("/content/drive/MyDrive/SRL_Symbols/loop_evolution_quantum.n
    print(f"✅ Quantum loop evolution regenerated: {len(slices)} slices"
else:
    print("⚠️ Not enough quantum loop windows for evolution tracking.")
```

```
✅ Quantum loop evolution regenerated: 1 slices
```

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# --- Load loop evolution matrices -
cosmic_evo = np.load("/content/drive
eeg_evo = np.load("/content/drive/My
quantum_evo = np.load("/content/driv

# --- Convert motifs to shared vocab
def build_vocab(*evolution_blocks):
    vocab = set()
    for block in evolution_blocks:
        for row in block:
            vocab.update(row)
    vocab = sorted(list(vocab))
    return vocab, {m: i for i, m in

vocab, vocab_idx = build_vocab(cosmi

# --- Encode sequences ---
def encode_sequence(seq, idx_map, vo
    vec = np.zeros(vocab_size)
    for motif in seq:
        vec[idx_map[motif]] += 1
    return vec / len(seq) if len(seq
```



```
# --- Compute cosine similarities ac
from sklearn.metrics.pairwise import

def compute_drift_matrix(evolution,
    return np.array([encode_sequence

cosmic_mat = compute_drift_matrix(co
eeg_mat = compute_drift_matrix(eeg_e
quantum_mat = compute_drift_matrix(q

# --- Compute convergence similarity
min_len = min(len(cosmic_mat), len(e
convergence = []
for i in range(min_len):
    sim1 = cosine_similarity([cosmic_
    sim2 = cosine_similarity([cosmic_
    sim3 = cosine_similarity([eeg_ma
    avg = (sim1 + sim2 + sim3) / 3
    convergence.append(avg)

# --- Plot ---
plt.figure(figsize=(10, 5))
plt.plot(convergence, marker='o')
plt.title("Spiral Loop Drift Converg
plt.xlabel("Time Window Index")
plt.ylabel("Average Cosine Similarit
plt.grid(True)
plt.tight_layout()
plt.show()
```