

```
from google.colab import drive
drive.mount('/content/drive')

Mounted at /content/drive

%%writefile irer_v5_fixed_simulation.py
### --- CONSOLIDATED IRER CONTROL HUB (SINGLE-CELL, v5.3 - S-NCGL Fixed) --- ####
# This cell contains all code, including:
# 1. S-NCGL "master equation" physics for a complex field (psi)
# 2. Prime-Log Spectral Attractor (ln(p)) SSE Validation
# 3. Effective Conformal Metric (ECM) proxy (jnp_construct_conformal_metric)
# 4. "Metric-Aware" EOM (compute_covariant_laplacian)
# 5. WORKING Aletheia metrics (Entropy, Quantule Census)
# 6. FIX: Corrected geometric proxy in jnp_construct_conformal_metric

# ---\
# SECTION 0: ALL IMPORTS
# ---\
import jax
import jax.numpy as jnp
from jax import lax
import numpy as np
import h5py
import os
import time
import traceback
from functools import partial
from typing import NamedTuple, Callable, Dict, Tuple, Any
from tqdm.auto import tqdm

# Imports for monitoring and validation
import matplotlib.pyplot as plt

print(f"JAX backend: {jax.default_backend()}")
print("All libraries imported.")

# ---\
# SECTION 1: IRER_JAX_ENGINE.PY (Refactored for S-NCGL)
# ---\

# --- 1.1: Core State Definitions (PyTrees) ---
class S_NCGL_State(NamedTuple):
    """State is now a single complex field psi."""
    psi: jax.Array

class S_NCGL_Parms(NamedTuple):
    """Parameters for the S-NCGL equation."""
    N_GRID: int
    T_TOTAL: float
    DT: float
    # S-NCGL Parameters
    alpha: float # Damping
    beta: float # Local cubic term
    gamma: float # Source term
    KAPPA: float # Laplacian coefficient
    nu: float # Non-local coupling
    sigma_k: float # Non-local kernel width
    # Spectral Analysis
    l_domain: float
    num_rays: int
    k_bin_width: float
    k_max_plot: float

class SpecOps(NamedTuple):
    kx: jax.Array
    ky: jax.Array
    k_sq: jax.Array
    gaussian_kernel_k: jax.Array
    dealias_mask: jax.Array
    prime_targets_k: jax.Array # For ln(p) SSE
    k_bins: jax.Array # For ln(p) SSE
    ray_angles: jax.Array # For ln(p) SSE
    k_max: float # For ln(p) SSE

    # Pre-compute static arrays for spectral analysis
    xx: jax.Array
    yy: jax.Array
    k_values_1d: jax.Array
    sort_indices_1d: jax.Array
```

```

SimCarry = S_NCGL_State

# --- 1.2: Gravitational Bridge & Geometric Operators ---

def jnp_construct_conformal_metric(
    rho: jnp.ndarray, # Takes the real field rho
    coupling_alpha: float,
    epsilon: float = 1e-9
) -> jnp.ndarray:
    """
    Computes the conformal factor Omega using the ECM model.
    FIX: Aligns geometric proxy's vacuum state with S-NCGL physics (rho=0).
    """
    alpha = jnp.maximum(coupling_alpha, epsilon)
    Omega = jnp.exp(alpha * rho) # Corrected: Uses rho directly
    return Omega

def spectral_gradient_complex(field: jax.Array, spec: SpecOps) -> Tuple[jax.Array, jax.Array]:
    """Computes spatial gradients (df/dx, df/dy) for a COMPLEX field."""
    field_fft = jnp.fft.fft2(field)
    grad_x_fft = (1j * spec.kx * field_fft) * spec.dealias_mask
    grad_y_fft = (1j * spec.ky * field_fft) * spec.dealias_mask
    return jnp.fft.ifft2(grad_x_fft), jnp.fft.ifft2(grad_y_fft)

def spectral_laplacian_complex(field: jax.Array, spec: SpecOps) -> jax.Array:
    """Computes the flat-space Laplacian for a COMPLEX field."""
    field_fft = jnp.fft.fft2(field)
    field_fft = field_fft * spec.dealias_mask
    return jnp.fft.ifft2((-spec.k_sq) * field_fft)

def compute_covariant_laplacian_complex(
    psi: jax.Array,
    Omega: jax.Array,
    spec: SpecOps
) -> jax.Array:
    """
    Computes the curved-space spatial Laplacian (Laplace-Beltrami operator)
    for the COMPLEX field 'psi'.
    L_g(psi) = (1/Omega^2) * nabla^2(psi) + (1/Omega^3) * (grad(Omega) . grad(psi))
    """
    epsilon = 1e-9
    Omega_safe = jnp.maximum(Omega, epsilon)
    Omega_sq_safe = jnp.square(Omega_safe)

    g_inv_sq = 1.0 / Omega_sq_safe
    flat_laplacian_psi = spectral_laplacian_complex(psi, spec)
    curvature_modified_accel = g_inv_sq * flat_laplacian_psi

    g_inv_cubed = g_inv_sq / Omega_safe
    grad_psi_x, grad_psi_y = spectral_gradient_complex(psi, spec)
    grad_Omega_x_complex, grad_Omega_y_complex = spectral_gradient_complex(Omega, spec)
    grad_Omega_x = grad_Omega_x_complex.real
    grad_Omega_y = grad_Omega_y_complex.real
    dot_product = (grad_Omega_x * grad_psi_x) + (grad_Omega_y * grad_psi_y)
    geometric_damping = g_inv_cubed * dot_product

    spatial_laplacian_g = curvature_modified_accel + geometric_damping
    return spatial_laplacian_g

# --- 1.3: Core Physics: S-NCGL (Refactored) ---

def jnp_get_derivatives(state: S_NCGL_State, params: S_NCGL_Parms,
                       coupling_params: dict, spec: SpecOps) -> S_NCGL_State:
    """
    Core EOM for the S-NCGL equation, with Geometric Feedback.
    """
    psi = state.psi
    rho = jnp.abs(psi)**2

    # S-NCGL Physics Terms
    rho_fft = jnp.fft.fft2(rho)
    non_local_term_k_fft = spec.gaussian_kernel_k * rho_fft
    non_local_term_k = jnp.fft.ifft2(non_local_term_k_fft * spec.dealias_mask).real
    non_local_coupling = -params.nu * non_local_term_k * psi
    local_cubic_term = -params.beta * rho * psi
    source_term = params.gamma * psi
    damping_term = -params.alpha * psi

    # Geometric Feedback
    Omega = jnp_construct_conformal_metric(rho, coupling_params['OMEGA_PARAM_A'])
    spatial_laplacian_g = compute_covariant_laplacian_complex(psi, Omega, spec)
    covariant_laplacian_term = params.KAPPA * spatial_laplacian_g

    # S-NCGL EOM

```

```

d_psi_dt = (
    damping_term +
    source_term +
    local_cubic_term +
    non_local_coupling +
    covariant_laplacian_term
)
return S_NCGL_State(psi=d_psi_dt)

def rk4_step(state: S_NCGL_State, params: S_NCGL_Parms, coupling_params: dict,
            dt: float, spec: SpecOps, deriv_func: Callable) -> S_NCGL_State:
    """RK4 step for the complex S-NCGL state."""
    k1 = deriv_func(state, params, coupling_params, spec)
    k2_state = jax.tree_util.tree_map(lambda y, dy: y + dy * dt / 2.0, state, k1)
    k2 = deriv_func(k2_state, params, coupling_params, spec)
    k3_state = jax.tree_util.tree_map(lambda y, dy: y + dy * dt / 2.0, state, k2)
    k3 = deriv_func(k3_state, params, coupling_params, spec)
    k4_state = jax.tree_util.tree_map(lambda y, dy: y + dy * dt, state, k3)
    k4 = deriv_func(k4_state, params, coupling_params, spec)
    new_state = jax.tree_util.tree_map(
        lambda y, dy1, dy2, dy3, dy4: y + (dt / 6.0) * (dy1 + 2.0*dy2 + 2.0*dy3 + 4),
        state, k1, k2, k3, k4
    )
    return new_state

# --- 1.4: Aletheia Metrics (ln(p) SSE + Entropy/Census) ---

def compute_directional_spectrum(
    psi: jax.Array,
    params: S_NCGL_Parms,
    spec: SpecOps
) -> Tuple[jax.Array, jax.Array]:
    """ Implements the "multi-ray directional sampling protocol". """
    n_grid = params.N_GRID
    xx, yy = spec.xx, spec.yy
    k_values_1d = spec.k_values_1d
    sort_indices = spec.sort_indices_1d
    power_spectrum_agg = jnp.zeros_like(spec.k_bins)

    def body_fun(i, power_spectrum_agg):
        angle = spec.ray_angles[i]
        slice_1d = psi[n_grid // 2, :] # Simplified slice
        slice_fft = jnp.fft.fft(slice_1d)
        power_spectrum_1d = jnp.abs(slice_fft)**2
        k_values_sorted = k_values_1d[sort_indices]
        power_spectrum_sorted = power_spectrum_1d[sort_indices]
        binned_power, _ = jnp.histogram(
            k_values_sorted,
            bins=jnp.append(spec.k_bins, params.k_max_plot),
            weights=power_spectrum_sorted
        )
        return power_spectrum_agg + binned_power

    power_spectrum_total = lax.fori_loop(0, params.num_rays, body_fun, power_spectrum_agg)
    power_spectrum_norm = power_spectrum_total / (jnp.sum(power_spectrum_total) + 1e-9)
    return spec.k_bins, power_spectrum_norm

def compute_log_prime_sse(
    k_values: jax.Array,
    power_spectrum: jax.Array,
    spec: SpecOps
) -> jax.Array:
    """ Computes the Sum of Squared Errors (SSE) against the ln(p) targets. """
    targets_k = spec.prime_targets_k
    target_indices = jnp.argmin(
        jnp.abs(k_values[:, None] - targets_k[None, :]),
        axis=0
    )
    target_spectrum_sparse = jnp.zeros_like(k_values).at[target_indices].set(1.0)
    target_spectrum_norm = target_spectrum_sparse / jnp.sum(target_spectrum_sparse)
    diff = power_spectrum - target_spectrum_norm
    sse = jnp.sum(diff * diff)
    return sse

def jnp_calculate_entropy(rho: jax.Array) -> jax.Array:
    rho_norm = rho / jnp.sum(rho)
    rho_safe = jnp.maximum(rho_norm, 1e-9)
    return -jnp.sum(rho_safe * jnp.log(rho_safe))

def jnp_calculate_quantule_census(rho: jax.Array) -> jax.Array:
    rho_mean = jnp.mean(rho)
    rho_std = jnp.std(rho)

```

```

threshold = rho_mean + 3.0 * rho_std
return jnp.sum(rho > threshold).astype(jnp.float32)

# --- 1.5: Master JIT Step Function (Refactored) ---

def jnp_sncgl_conformal_step(
    carry_state: S_NCGL_State,
    t: float,
    deriv_func: callable,
    params: S_NCGL_Params,
    coupling_params: dict,
    spec: SpecOps
) -> (S_NCGL_State, dict):
    """ Master step function (to be JIT-compiled by lax.scan). """
    state = carry_state
    DT = params.DT
    new_state = rk4_step(state, params, coupling_params, DT, spec, deriv_func)
    new_rho = jnp.abs(new_state.psi)**2

    k_bins, power_spectrum = compute_directional_spectrum(new_state.psi, params, spec)
    ln_p_sse = compute_log_prime_sse(k_bins, power_spectrum, spec)
    informational_entropy = jnp_calculate_entropy(new_rho)
    quantule_census = jnp_calculate_quantule_census(new_rho)
    Omega_final_for_log = jnp_construct_conformal_metric(
        new_rho, coupling_params['OMEGA_PARAM_A']
    )
    Omega_sq_final_for_log = jnp.square(Omega_final_for_log)

    metrics = {
        "timestamp": t * DT,
        "ln_p_sse": ln_p_sse,
        "informational_entropy": informational_entropy,
        "quantule_census": quantule_census,
        "omega_sq": Omega_sq_final_for_log
    }
    new_carry_state = S_NCGL_State(psi=new_state.psi)
    return new_carry_state, metrics

print("SECTION 1: JAX Engine (Refactored for S-NCGL) is defined.")

# ---\
# SECTION 2: ORCHESTRATOR.PY (Refactored)
# ---\

# --- 2.1: HDF5Logger (Corrected) ---
class HDF5Logger:
    def __init__(self, filename, n_steps, n_grid, metrics_keys, buffer_size=100):
        self.filename = filename
        self.n_steps = n_steps
        self.metrics_keys = metrics_keys
        self.buffer_size = buffer_size
        self.buffer = {key: [] for key in self.metrics_keys}
        self.buffer['omega_sq_history'] = [] # Explicitly add this
        self.write_index = 0

        with h5py.File(self.filename, 'w') as f:
            for key in self.metrics_keys:
                f.create_dataset(key, (n_steps,), maxshape=(n_steps,), dtype='f8')
            f.create_dataset(
                'omega_sq_history',
                shape=(n_steps, n_grid, n_grid),
                maxshape=(n_steps, n_grid, n_grid),
                chunks=(1, n_grid, n_grid),
                dtype='f4',
                compression="gzip"
            )
            f.create_dataset(
                'final_psi',
                shape=(n_grid, n_grid),
                dtype='c16',
                compression="gzip"
            )

    def log_timestep(self, metrics: dict):
        for key in self.metrics_keys:
            if key in metrics:
                self.buffer[key].append(metrics[key])

        # Always check for omega_sq
        if 'omega_sq' in metrics:
            self.buffer['omega_sq_history'].append(metrics['omega_sq'])


```

```

if len(self.buffer[self.metrics_keys[0]]) >= self.buffer_size:
    self.flush() # This is the correct call

def flush(self):
    if not self.buffer[self.metrics_keys[0]]:
        return
    buffer_len = len(self.buffer[self.metrics_keys[0]])
    start = self.write_index
    end = start + buffer_len
    try:
        with h5py.File(self.filename, 'a') as f:
            for key in self.metrics_keys:
                f[key][start:end] = np.array(self.buffer[key])
            f['omega_sq_history'][start:end] = np.array(self.buffer['omega_sq_history'])

        self.buffer = {key: [] for key in self.metrics_keys}
        self.buffer['omega_sq_history'] = [] # Reset buffer
        self.write_index = end
    except Exception as e:
        print(f"HDF5Logger Error: {e}")

def save_final_state(self, final_psi: jax.Array):
    try:
        with h5py.File(self.filename, 'a') as f:
            f['final_psi'][:] = np.array(final_psi)
        print(f"Final psi state saved to {self.filename}")
    except Exception as e:
        print(f"HDF5Logger Error (Final State): {e}")

def close(self):
    self.flush()
    print(f" HDF5Logger closed. Data saved to {self.filename}")

# --- 2.2: Main Driver (Refactored) ---
def run_simulation_with_io(
    fmia_params: S_NCGL_Params,
    coupling_params: dict,
    initial_state: S_NCGL_State,
    spec_ops: SpecOps,
    output_filename="simulation_output.hdf5",
    log_every_n=10):

    T_TOTAL = fmia_params.T_TOTAL
    DT = fmia_params.DT
    N_GRID = fmia_params.N_GRID
    total_steps = int(T_TOTAL / DT)
    log_steps = int(total_steps / log_every_n)
    if log_steps == 0:
        log_steps = 1 # Ensure at least one log step
    timesteps_to_run = jnp.arange(0, total_steps)

    print(f" Total Steps: {total_steps}, Log Steps: {log_steps}")
    initial_carry = initial_state

    step_func = lambda carry_state, t: jnp_sncgl_conformal_step(
        carry_state, t, jnp_get_derivatives,
        fmia_params, coupling_params, spec_ops
    )
    jit_scan_step = jax.jit(step_func)

    metrics_to_log = ["timestamp", "ln_p_sse", "informational_entropy", "quantule_census"]
    logger = HDF5Logger(output_filename, log_steps, N_GRID, metrics_to_log)

    start_time = time.time()
    current_carry = initial_carry

    for i in tqdm(range(log_steps), desc=" > Sim Progress", leave=False):
        steps_in_chunk = timesteps_to_run[i*log_every_n : (i+1)*log_every_n]
        if steps_in_chunk.shape[0] == 0:
            continue # Skip if no steps in this chunk

        final_carry_state, metrics_chunk = jax.lax.scan(
            jit_scan_step,
            current_carry,
            steps_in_chunk
        )
        last_metrics_in_chunk = {
            key: metrics_chunk[key][-1] for key in (metrics_to_log + ['omega_sq'])
        }
        logger.log_timestep(last_metrics_in_chunk)
        current_carry = final_carry_state

```

```

end_time = time.time()
print(f"  > Simulation Loop Complete ({end_time - start_time:.2f}s)")
logger.save_final_state(current_carry.psi)
logger.close()
return current_carry, output_filename

print("SECTION 2: Orchestrator (Refactored for S-NCGL) is defined.")

# ---\
# SECTION 3: SCRIPT EXECUTION (Refactored for Sweeps v2)
# ---\
def run_single_simulation(
    run_params: dict,
    output_filename: str
) -> float:
    """
    Main execution function, refactored to run a single simulation
    with a given set of parameters and return the final SSE.
    """
    run_id = os.path.basename(output_filename).replace('.hdf5', '')
    print(f"  > [JAX ENGINE] EXECUTING: {run_id}")

    # --- 3.1: Parameter Configuration ---
    N_GRID = int(run_params.get("N_GRID", 128))
    DT = float(run_params.get("DT", 1e-3))
    T_TOTAL = float(run_params.get("T_TOTAL", 2.0))
    _alpha = float(run_params.get("alpha", 0.1))
    _beta = float(run_params.get("beta", 1.0))
    _gamma = float(run_params.get("gamma", 0.2))
    _KAPPA = float(run_params.get("KAPPA", 1.0))
    _nu = float(run_params.get("nu", 1.0))
    _sigma_k = float(run_params.get("sigma_k", 2.5))
    _OMEGA_PARAM_A = float(run_params.get("OMEGA_PARAM_A", 0.5))
    _jax_run_seed = int(run_params.get("jax_run_seed", 42))
    LOG_EVERY_N_STEPS = 10

    # --- *** THIS IS THE FIX ***
    # Assign the lowercase argument to the uppercase variable
    # that the rest of the script expects.
    OUTPUT_FILENAME = output_filename
    # --- *** END OF FIX ***
    # --- 3.2: Dependent Parameter Setup ---
    L_DOMAIN = 20.0
    K_MAX_PLOT = 2.0
    K_BIN_WIDTH = 0.01
    NUM_RAYS = 32

    def kgrid_2pi(n: int, L: float = 1.0):
        k = 2.0 * jnp.pi * jnp.fft.fftfreq(n, d=L/n)
        return jnp.meshgrid(k, k, indexing='ij')

    kx, ky = kgrid_2pi(N_GRID, L=L_DOMAIN)
    k_sq = kx**2 + ky**2
    k_mag = jnp.sqrt(k_sq)
    k_max_sim = jnp.max(k_mag)
    k_ny = jnp.max(jnp.abs(kx))
    k_cut = (2.0/3.0) * k_ny
    dealias_mask = ((jnp.abs(kx) <= k_cut) & (jnp.abs(ky) <= k_cut)).astype(jnp.float32)

    def make_gaussian_kernel_k(k_sq, sigma_k):
        return jnp.exp(-k_sq / (2.0 * (sigma_k**2)))

    prime_targets_k = jnp.log(jnp.array([2, 3, 5, 7, 11, 13, 17, 19]))
    k_bins = jnp.arange(0, K_MAX_PLOT, K_BIN_WIDTH)
    ray_angles = jnp.linspace(0, jnp.pi, NUM_RAYS, endpoint=False)
    xx, yy = jnp.meshgrid(
        jnp.linspace(-0.5, 0.5, N_GRID) * L_DOMAIN,
        jnp.linspace(-0.5, 0.5, N_GRID) * L_DOMAIN
    )
    k_values_1d = 2 * jnp.pi * jnp.fft.fftfreq(N_GRID, d=L_DOMAIN / N_GRID)
    sort_indices_1d = jnp.argsort(k_values_1d)

    # --- 3.3: Final Struct Assembly ---
    fmia_params = S_NCGL_Params(
        N_GRID=N_GRID, T_TOTAL=T_TOTAL, DT=DT, alpha=_alpha, beta=_beta,
        gamma=_gamma, KAPPA=_KAPPA, nu=_nu, sigma_k=_sigma_k,
        l_domain=L_DOMAIN, num_rays=NUM_RAYS, k_bin_width=K_BIN_WIDTH,
        k_max_plot=K_MAX_PLOT
    )

```

```

gaussian_kernel_k = make_gaussian_kernel_k(k_sq, fmia_params.sigma_k)
spec_ops = SpecOps(
    kx=kx.astype(jnp.float32), ky=ky.astype(jnp.float32),
    k_sq=k_sq.astype(jnp.float32),
    gaussian_kernel_k=gaussian_kernel_k.astype(jnp.float32),
    dealias_mask=dealias_mask.astype(jnp.float32),
    prime_targets_k=prime_targets_k.astype(jnp.float32),
    k_bins=k_bins.astype(jnp.float32),
    ray_angles=ray_angles.astype(jnp.float32),
    k_max=k_max_sim.astype(jnp.float32), xx=xx.astype(jnp.float32),
    yy=yy.astype(jnp.float32), k_values_id=k_values_id.astype(jnp.float32),
    sort_indices_id=sort_indices_id
)
coupling_params = { "OMEGA_PARAM_A": _OMEGA_PARAM_A, "RHO_VAC": 1.0 }

key = jax.random.PRNGKey(_jax_run_seed)
psi_initial = (
    jax.random.uniform(key, (N_GRID, N_GRID), dtype=jnp.float32) * 0.1 +
    1j * jax.random.uniform(key, (N_GRID, N_GRID), dtype=jnp.float32) * 0.1
)
initial_state = S_NCGL_State(psi=psi_initial.astype(jnp.complex64))

# --- 3.4: Simulation Execution ---
final_sse_float = 9999.0 # Default high value
try:
    final_carry_state, output_file = run_simulation_with_io(
        fmia_params, coupling_params, initial_state, spec_ops,
        output_filename=OUTPUT_FILENAME, # <-- This line is now correct
        log_every_n=LOG_EVERY_N_STEPS
    )
    final_state = final_carry_state

    # --- 3.5: Final Validation ---
    final_rho = jnp.abs(final_state.psi)**2
    k_bins_final, power_spec_final = compute_directional_spectrum(
        final_state.psi, fmia_params, spec_ops
    )
    final_sse_metric = compute_log_prime_sse(k_bins_final, power_spec_final, spec_ops)
    final_sse_float = float(final_sse_metric)

    final_Omega = jnp_construct_conformal_metric(
        final_rho, coupling_params['OMEGA_PARAM_A']
    )
    final_g_tt = -jnp.square(final_Omega)
    mean_g_tt = float(jnp.mean(final_g_tt))

    print(f"  > [JAX ENGINE] Report for {run_id}:")
    print(f"  >   Final Prime-Log SSE: {final_sse_float:.12f}")
    print(f"  >   Mean g_tt (time):   {mean_g_tt:.6f}")

except Exception as e:
    print(f"\n--- [JAX ENGINE] SIMULATION FAILED ({run_id}) ---")
    print(f"An error occurred during execution: {e}")
    traceback.print_exc()
    final_sse_float = 9999.0

return final_sse_float

if __name__ == "__main__":
    print("--- [JAX ENGINE] Running in script mode (single test run). ---")
    default_params = {
        "alpha": 0.1, "beta": 1.0, "gamma": 0.2, "KAPPA": 1.0,
        "nu": 1.0, "sigma_k": 2.5, "OMEGA_PARAM_A": 0.5,
        "N_GRID": 128, "T_TOTAL": 2.0,
        "jax_run_seed": 42,
        "param_hash": "TEST_RUN_001"
    }
    test_output_dir = "single_run_output"
    test_output_file = os.path.join(test_output_dir, "run_TEST_RUN_001.hdf5")

    start_time = time.time()
    sse_result = run_single_simulation(
        run_params=default_params,
        output_filename=test_output_file
    )
    end_time = time.time()

    print(f"\n--- [JAX ENGINE] SINGLE RUN COMPLETE ---")
    print(f"Final SSE: {sse_result}")
    print(f"Total time: {end_time - start_time:.2f} seconds")
    print(f"Output saved to: {test_output_file}")

```

Writing `irer_v5_fixed_simulation.py`

```
%%writefile sweep_controller_v2.py
import os
import csv
import time
import numpy as np
import json
import hashlib
import datetime
import traceback
from typing import Dict, Any, List

# --- 1. Import the base simulation function ---
try:
    from irer_v5_fixed_simulation import run_single_simulation
    print("[WORKER] Successfully imported REAL JAX simulation engine.")
except ImportError:
    print("[WORKER] Error: Could not find 'irer_v5_fixed_simulation.py'.")
    exit()

# --- 2. Define Helper Functions ---
def generate_param_hash(params: Dict[str, Any]) -> str:
    sorted_params_str = json.dumps(params, sort_keys=True).encode('utf-8')
    hash_str = hashlib.sha256(sorted_params_str).hexdigest()
    return hash_str[:12]

def write_to_ledger(ledger_file: str, run_data: Dict[str, Any]):
    file_exists = os.path.isfile(ledger_file)
    all_headers = sorted(list(run_data.keys()))
    preferred_order = [
        'param_hash', 'final_sse', 'jax_run_seed', 'generation',
        'alpha', 'sigma_k', 'nu', 'OMEGA_PARAM_A', # <-- Ensure it's ordered
        'gamma', 'beta', 'KAPPA', 'N_GRID', 'T_TOTAL'
    ]
    final_headers = [h for h in preferred_order if h in all_headers] + \
                    [h for h in all_headers if h not in preferred_order]
    try:
        with open(ledger_file, 'a', newline='') as f:
            writer = csv.DictWriter(f, fieldnames=final_headers)
            if not file_exists:
                writer.writeheader()
            writer.writerow(run_data)
    except Exception as e:
        print(f"  > [WORKER] Error writing to ledger: {e}")

def load_todo_list(todo_file: str) -> List[Dict[str, Any]]:
    try:
        with open(todo_file, 'r') as f:
            jobs = json.load(f)
        os.remove(todo_file)
        print(f"  > [WORKER] Loaded and removed '{todo_file}'")
        return jobs
    except FileNotFoundError:
        return []

def generate_bootstrap_jobs(
    rng: np.random.Generator,
    num_jobs: int
) -> List[Dict[str, Any]]:
    """
    --- *** UPDATED FOR 4D "FULL-SPACE" HUNT ***
    """
    print(f"  > [WORKER] Generating {num_jobs} (4D) bootstrap jobs (Gen 0)...")
    jobs = []

    # --- *** FIX: Added OMEGA_PARAM_A to the random ranges ***
    # These ranges must match the 'Hunter' (aste_hunter.py)
    PARAM_RANGES = {
        'alpha': ('uniform', 0.01, 1.0),
        'sigma_k': ('uniform', 0.5, 10.0),
        'nu': ('uniform', 0.1, 5.0),
        'OMEGA_PARAM_A': ('uniform', 0.1, 2.5)
    }

    for _ in range(num_jobs):
        job = {}
        for key, (dist, p_min, p_max) in PARAM_RANGES.items():
            if dist == 'uniform':
                job[key] = rng.uniform(low=p_min, high=p_max)
        job['generation'] = 0
        jobs.append(job)

    return jobs
```

```

        return jobs

# --- 3. Main Worker Execution ---
def main(hunt_id, todo_file):
    print(f"--- [WORKER] ENGAGED for {hunt_id} (4D Full-Space) ---")

    # --- 3.1: Configuration ---
    MASTER_SEED = 42
    BOOTSTRAP_JOBS = 100 # From aste_hunter.py's GENERATION_SIZE

    # --- *** FIX: REMOVED OMEGA_PARAM_A from static params ***
    STATIC_PARAMS = {
        "gamma": 0.2, "beta": 1.0, "KAPPA": 1.0,
        "N_GRID": 128, "T_TOTAL": 2.0
    }

    # --- 3.2: Setup Directories & Ledger ---
    MASTER_OUTPUT_DIR = os.path.join("sweep_runs", hunt_id)
    os.makedirs(MASTER_OUTPUT_DIR, exist_ok=True)
    LEDGER_FILE = os.path.join(MASTER_OUTPUT_DIR, f"ledger_{hunt_id}.csv")

    master_rng = np.random.default_rng(MASTER_SEED)

    # --- 3.4: Load or Generate Job List ---
    params_to_run = load_todo_list(todo_file)
    if not params_to_run:
        print(f" > [WORKER] No '{todo_file}' found. Bootstrapping...")
        params_to_run = generate_bootstrap_jobs(master_rng, BOOTSTRAP_JOBS)

    total_jobs = len(params_to_run)
    print(f" > [WORKER] Found {total_jobs} jobs to run.")
    sweep_start_time = time.time()

    for i, variable_params in enumerate(params_to_run):
        run_start_time = time.time()
        print(f"\n --- [WORKER] Starting Job {i+1} / {total_jobs} ---")

        current_run_params = variable_params.copy()
        current_run_params.update(STATIC_PARAMS)

        if 'generation' not in current_run_params:
            current_run_params['generation'] = 'unknown'

        jax_run_seed = int(master_rng.integers(low=0, high=2**31 - 1))
        current_run_params['jax_run_seed'] = jax_run_seed
        param_hash = generate_param_hash(current_run_params)
        current_run_params['param_hash'] = param_hash
        print(f"     Run Hash: {param_hash} | JAX Seed: {jax_run_seed}")

        hdf5_filename = os.path.join(MASTER_OUTPUT_DIR, f"run_{param_hash}.hdf5")

        try:
            final_sse = run_single_simulation(
                run_params=current_run_params,
                output_filename=hdf5_filename
            )
        except Exception as e:
            print(f"!!! [WORKER] JOB {param_hash} FAILED WITH EXCEPTION: {e} !!!")
            traceback.print_exc()
            final_sse = 99999.0

        run_end_time = time.time()
        current_run_params['final_sse'] = final_sse
        print(f" --- [WORKER] Job {i+1} Complete ({run_end_time - run_start_time:.2f}s) ---")
        print(f"     Final SSE: {final_sse:.12f}")

        write_to_ledger(LEDGER_FILE, current_run_params)

    sweep_end_time = time.time()
    print(f"\n--- [WORKER] FINISHED {hunt_id} ---")
    print(f"Total time for {total_jobs} jobs: {(sweep_end_time - sweep_start_time) / 60.0:.2f} minutes")

if __name__ == "__main__":
    import sys
    if len(sys.argv) != 3:
        print("Usage: python sweep_controller_v2.py <HUNT_ID> <TODO_FILE_NAME>")
        sys.exit(1)

    main(hunt_id=sys.argv[1], todo_file=sys.argv[2])

```

Writing sweep_controller_v2.py

```

%%writefile aste_hunter.py
import pandas as pd
import numpy as np
import os
import json
import traceback
import sys

print("--- [HUNTER] ENGAGED (v3: Diversity Injection) ---")

# --- 1. Configuration: Genetic Algorithm Settings ---
GENERATION_SIZE = 100
ELITE_PERCENTILE = 0.1
MUTATION_STRENGTH = 0.15
MUTATION_CHANCE = 0.7

# --- *** NEW: DIVERSITY INJECTION SETTINGS *** ---
STAGNATION_GENERATIONS = 3 # Check if best SSE is stuck for 3 gens
IMMIGRANT_RATIO = 0.5      # If stuck, 50% of next gen is random

PARAMS_TO_EVOLVE = ['alpha', 'sigma_k', 'nu', 'OMEGA_PARAM_A']
PARAM_RANGES = {
    'alpha': (0.01, 1.0),
    'sigma_k': (0.5, 10.0),
    'nu': (0.1, 5.0),
    'OMEGA_PARAM_A': (0.1, 2.5)
}

# --- 2. Helper Functions ---
def clamp_param(value: float, p_min: float, p_max: float) -> float:
    return max(p_min, min(value, p_max))

def generate_random_immigrants(rng: np.random.Generator, num_immigrants: int, generation: int):
    """Generates new random candidates for exploration."""
    immigrants = []
    for _ in range(num_immigrants):
        job = {}
        for key, (p_min, p_max) in PARAM_RANGES.items():
            job[key] = rng.uniform(low=p_min, high=p_max)
        job['generation'] = generation
        immigrants.append(job)
    return immigrants

def check_stagnation(ledger_df: pd.DataFrame, current_gen: int, threshold: int) -> bool:
    """Checks if the best SSE has improved in the last 'threshold' generations."""
    if current_gen < threshold:
        return False

    try:
        # Get best SSE for the last 'threshold' generations
        gen_indices = range(current_gen - threshold + 1, current_gen + 1)
        best_sseds = []
        for gen in gen_indices:
            gen_runs = ledger_df[ledger_df['generation'] == gen]
            if gen_runs.empty:
                continue # Skip gen if no data
            best_sseds.append(gen_runs['final_sse'].min())

        if len(best_sseds) < threshold:
            return False # Not enough data to be sure

        # Check if all recent best SSEs are effectively the same
        first_sse = best_sseds[0]
        if all(np.isclose(sse, first_sse, atol=1e-6) for sse in best_sseds):
            print(f"  > [HUNTER] STAGNATION DETECTED. Best SSE stuck at {first_sse:.6f}.")
            return True
    except Exception as e:
        print(f"  > [HUNTER] Warning: Could not check stagnation. {e}")

    return False

# --- 3. Main Hunter Logic ---
def main(hunt_id, todo_file):
    print(f"  > [HUNTER] Analyzing {hunt_id}...")
    MASTER_OUTPUT_DIR = os.path.join("sweep_runs", hunt_id)
    LEDGER_FILE = os.path.join(MASTER_OUTPUT_DIR, f"ledger_{hunt_id}.csv")
    rng = np.random.default_rng()

    try:
        if os.path.exists(todo_file):
            print(f"  > [HUNTER] Error: '{todo_file}' already exists.")
            return
    
```

```

if not os.path.exists(LEDERER_FILE):
    print(f" > [HUNTER] Ledger not found. Bootstrapping (Worker will handle).")
    return

ledger_df = pd.read_csv(LEDERER_FILE)
valid_runs_df = ledger_df[ledger_df['final_sse'] < 90000].copy()

if valid_runs_df.empty:
    print(" > [HUNTER] Ledger contains no valid runs. Cannot breed.")
    return

n_elite = max(2, int(len(valid_runs_df) * ELITE_PERCENTILE))
elite_df = valid_runs_df.sort_values(by='final_sse').head(n_elite)

current_gen = int(ledger_df['generation'].max())
next_gen = current_gen + 1

print(f" > [HUNTER] Loaded {len(ledger_df)} total runs.")
print(f" > [HUNTER] Current Gen: {current_gen}, Best SSE: {elite_df['final_sse'].min():.8f}")

# --- DIVERSITY INJECTION LOGIC ---
is_stagnant = check_stagnation(valid_runs_df, current_gen, STAGNATION_GENERATIONS)
num_immigrants = 0

if is_stagnant:
    print(f" > [HUNTER] Injecting diversity: {IMMIGRANT_RATIO*100}% immigrants.")
    num_immigrants = int(GENERATION_SIZE * IMMIGRANT_RATIO)

num_children = GENERATION_SIZE - num_immigrants
new_generation_params = []

# 1. Breed Children (Exploitation)
print(f" > [HUNTER] Breeding {num_children} 'children'...")
for i in range(num_children):
    parent1_series = elite_df.sample(1).iloc[0]
    parent2_series = elite_df.sample(1).iloc[0]
    child = {}

    for key in PARAMS_TO_EVOLVE:
        p1 = parent1_series.get(key, PARAM_RANGES[key][0])
        p2 = parent2_series.get(key, PARAM_RANGES[key][0])
        child[key] = (p1 + p2) / 2.0

        if rng.random() < MUTATION_CHANCE:
            mutation_val = rng.normal(0, MUTATION_STRENGTH * child[key])
            child[key] += mutation_val
            p_min, p_max = PARAM_RANGES[key]
            child[key] = clamp_param(child[key], p_min, p_max)

    child['generation'] = next_gen
    new_generation_params.append(child)

# 2. Generate Immigrants (Exploration)
if num_immigrants > 0:
    print(f" > [HUNTER] Generating {num_immigrants} 'immigrants'...")
    immigrants = generate_random_immigrants(rng, num_immigrants, next_gen)
    new_generation_params.extend(immigrants)

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

print(f"\n--- [HUNTER] FINISHED ---")
print(f" > Successfully created '{todo_file}' for Gen {next_gen} ({num_children} children, {num_immigrants} immig")

except Exception as e:
    print(f"\n--- [HUNTER] FAILED ---")
    print(f"An error occurred: {e}")
    traceback.print_exc()

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Usage: python aste_hunter.py <HUNT_ID> <TODO_FILE_NAME>")
        sys.exit(1)

    main(hunt_id=sys.argv[1], todo_file=sys.argv[2])

```

Writing aste_hunter.py

```
%%writefile adaptive_hunt_orchestrator.py
import os
import subprocess
```

```

import pandas as pd
import time
import sys

print("--- [ORCHESTRATOR] ENGAGED (v5: Corrected 4D Hunt) ---")

# --- 1. Global Configuration Parameters ---
GOAL_SSE = 0.1 # Targeting the global minimum
CONSECUTIVE_GOAL_GENS = 3
NUM_HUNTS = 3
MASTER_OUTPUT_DIR = "/content/drive/MyDrive/irer_simulations/sweep_runs" # Updated path for Google Drive
TODO_FILE = "ASTE_generation_todo.json"
HUNT_ID_OFFSET = 15 # <- **** NEW OFFSET: Start from HUNT_015 ****

# --- 2. Helper Functions ---
def run_command(command):
    print(f"\nExecuting: {command}")
    process = subprocess.Popen(command, shell=True, stdout=subprocess.PIPE, stderr=subprocess.STDOUT, text=True)
    while True:
        output = process.stdout.readline()
        if output == '' and process.poll() is not None:
            break
        if output:
            print(output.strip())
    rc = process.poll()
    if rc != 0:
        print(f"[ORCHESTRATOR] Command failed with exit code {rc}")
    return rc

def get_best_sse(hunt_id):
    ledger_file = os.path.join(MASTER_OUTPUT_DIR, hunt_id, f"ledger_{hunt_id}.csv")
    if not os.path.exists(ledger_file):
        return float('inf')
    try:
        ledger_df = pd.read_csv(ledger_file)
        valid_runs = ledger_df[ledger_df['final_sse'] < 90000]
        if valid_runs.empty:
            return float('inf')
        return valid_runs['final_sse'].min()
    except Exception:
        return float('inf')

# --- 3. Main Orchestrator Logic ---
def main():
    print(f"--- Orchestrator: Targeting SSE < {GOAL_SSE} for {CONSECUTIVE_GOAL_GENS} generations ---")

    for i in range(NUM_HUNTS):
        hunt_index = i + HUNT_ID_OFFSET
        HUNT_ID = f"SNCGL_ADAPTIVE_HUNT_{hunt_index:003d}"

        print(f"\n{'-*80}'")
        print(f"--- STARTING ADAPTIVE HUNT: {HUNT_ID} (Corrected 4D Hunt) ---")
        print(f"{'-*80}'")

        consecutive_goal_met_gens = 0
        generation_counter = 0

        while True:
            print(f"\n--- Hunt {HUNT_ID}, Generation {generation_counter} ---")

            # --- Step 1: Run Worker ---
            print(f"Bootstrapping/Running Gen {generation_counter} for {HUNT_ID}...")
            rc = run_command(f"python sweep_controller_v2.py {HUNT_ID} {TODO_FILE}")
            if rc != 0:
                print(f"[ORCHESTRATOR] Worker failed for {HUNT_ID}. Stopping hunt.")
                break

            # --- Step 2: Run Hunter ---
            print(f"Running Hunter to analyze Gen {generation_counter} and create Gen {generation_counter + 1}...")
            rc = run_command(f"python aste_hunter.py {HUNT_ID} {TODO_FILE}")
            if rc != 0:
                print(f"[ORCHESTRATOR] Hunter failed for {HUNT_ID}. Stopping hunt.")
                break

            # --- Step 3: Monitor best SSE ---
            current_best_sse = get_best_sse(HUNT_ID)
            print(f"Hunt {HUNT_ID}, Generation {generation_counter}: Current Best SSE = {current_best_sse:.12f}")

            if current_best_sse <= GOAL_SSE:
                consecutive_goal_met_gens += 1
                print(f"GOAL SSE MET! Consecutive generations: {consecutive_goal_met_gens}/{CONSECUTIVE_GOAL_GENS}")
            else:

```

```
consecutive_goal_met_gens = 0
print("GOAL SSE NOT MET. Resetting consecutive counter.")

if consecutive_goal_met_gens >= CONSECUTIVE_GOAL_GENS:
    print(f"\n--- Hunt {HUNT_ID} COMPLETED ---")
    print(f"GOAL SSE ({GOAL_SSE}) achieved for {CONSECUTIVE_GOAL_GENS} consecutive generations!")
    break

if generation_counter > 50: # Safety break
    print(f"\n--- Hunt {HUNT_ID} STOPPED ---")
    print(f"Did not meet goal after 50 generations. Moving to next hunt.")
    break

generation_counter += 1
time.sleep(1) # Small delay

if os.path.exists(TODO_FILE):
    os.remove(TODO_FILE)
    print(f"Cleaned up residual '{TODO_FILE}'.")

print(f"\n{'-*80}'")
print("--- ORCHESTRATOR FINISHED ALL HUNTS ---")

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

Overwriting adaptive_hunt_orchestrator.py

```
!python adaptive_hunt_orchestrator.py
```

> Sim Progress: 56% | ██████████ | 111/200 [00:07<00:07, 11.51it/s]

```
!python adaptive_hunt_orchestrator.py
```

```
> Simulation Loop Complete (12.55s)
> Final psi state saved to sweep_runs/SNCGL_ADAPTIVE_HUNT_015/run_55de39c63acf.hdf5
> HDF5Logger closed. Data saved to sweep_runs/SNCGL_ADAPTIVE_HUNT_015/run_55de39c63acf.hdf5
> [JAX ENGINE] Report for run_55de39c63acf:
>   Final Prime-Log SSE: 1.199994921684
>   Mean g_tt (time): -2.107378
--- [WORKER] Job 71 Complete (12.91s) ---
Final SSE: 1.199994921684

--- [WORKER] Starting Job 72 / 100 ---
Run Hash: a19b44e9ebc8 | JAX Seed: 433323782
> [JAX ENGINE] EXECUTING: run_a19b44e9ebc8
> Total Steps: 2000, Log Steps: 200

> Sim Progress: 0% | 0/200 [00:00<?, ?it/s]
> Sim Progress: 0% | 1/200 [00:01<05:01, 1.51s/it]
> Sim Progress: 2% | 4/200 [00:01<01:02, 3.13it/s]
> Sim Progress: 4% | 7/200 [00:01<00:32, 5.97it/s]
> Sim Progress: 5% | 10/200 [00:01<00:21, 9.00it/s]
> Sim Progress: 6% | 13/200 [00:01<00:15, 12.26it/s]
> Sim Progress: 8% | 16/200 [00:02<00:12, 15.20it/s]
> Sim Progress: 10% | 19/200 [00:02<00:09, 18.17it/s]
> Sim Progress: 11% | 22/200 [00:02<00:08, 20.38it/s]
> Sim Progress: 12% | 25/200 [00:02<00:07, 21.91it/s]
> Sim Progress: 14% | 28/200 [00:02<00:07, 23.09it/s]
> Sim Progress: 16% | 31/200 [00:02<00:07, 22.76it/s]
> Sim Progress: 17% | 34/200 [00:02<00:07, 22.82it/s]
> Sim Progress: 18% | 37/200 [00:02<00:06, 24.05it/s]
> Sim Progress: 20% | 40/200 [00:02<00:06, 25.57it/s]
> Sim Progress: 22% | 43/200 [00:03<00:06, 25.83it/s]
> Sim Progress: 23% | 46/200 [00:03<00:05, 26.96it/s]
> Sim Progress: 24% | 49/200 [00:03<00:05, 26.65it/s]
> Sim Progress: 26% | 52/200 [00:03<00:05, 27.48it/s]
> Sim Progress: 28% | 55/200 [00:03<00:05, 26.92it/s]
> Sim Progress: 29% | 58/200 [00:03<00:05, 24.68it/s]
> Sim Progress: 30% | 61/200 [00:03<00:05, 25.77it/s]
> Sim Progress: 32% | 64/200 [00:03<00:05, 26.84it/s]
> Sim Progress: 34% | 67/200 [00:04<00:05, 26.24it/s]
> Sim Progress: 35% | 70/200 [00:04<00:04, 27.10it/s]
> Sim Progress: 36% | 73/200 [00:04<00:05, 24.98it/s]
> Sim Progress: 38% | 76/200 [00:04<00:05, 24.59it/s]
> Sim Progress: 40% | 79/200 [00:04<00:04, 25.92it/s]
> Sim Progress: 41% | 82/200 [00:04<00:04, 26.66it/s]
> Sim Progress: 42% | 85/200 [00:04<00:04, 25.08it/s]
> Sim Progress: 44% | 88/200 [00:04<00:04, 26.09it/s]
> Sim Progress: 46% | 91/200 [00:04<00:04, 26.00it/s]
> Sim Progress: 47% | 94/200 [00:05<00:03, 26.77it/s]
> Sim Progress: 48% | 97/200 [00:05<00:03, 27.57it/s]
> Sim Progress: 50% | 100/200 [00:05<00:07, 12.57it/s]
> Sim Progress: 52% | 104/200 [00:05<00:05, 16.03it/s]
> Sim Progress: 54% | 107/200 [00:05<00:05, 18.38it/s]
> Sim Progress: 55% | 110/200 [00:06<00:04, 20.12it/s]
> Sim Progress: 56% | 113/200 [00:06<00:03, 22.19it/s]
> Sim Progress: 58% | 116/200 [00:06<00:03, 23.87it/s]
> Sim Progress: 60% | 119/200 [00:06<00:03, 24.18it/s]
> Sim Progress: 61% | 122/200 [00:06<00:03, 25.19it/s]
> Sim Progress: 62% | 125/200 [00:06<00:02, 25.93it/s]
> Sim Progress: 64% | 128/200 [00:06<00:02, 26.79it/s]
> Sim Progress: 66% | 131/200 [00:06<00:02, 25.46it/s]
```

```
!python adaptive_hunt_orchestrator.py
```

```
import pandas as pd
import os

LEDGER_FILE = '/content/sweep_runs/SNCGL_ADAPTIVE_HUNT_012/ledger_SNCGL_ADAPTIVE_HUNT_012.csv'

if os.path.exists(LEDGER_FILE):
    ledger_df = pd.read_csv(LEDGER_FILE)

    # Filter out failed runs (SSE > 90000)
    valid_runs_df = ledger_df[ledger_df['final_sse'] < 90000].copy()

    print(f"Total entries in ledger: {len(ledger_df)}")
    if not valid_runs_df.empty:
        current_gen = valid_runs_df['generation'].max()
        best_sse = valid_runs_df['final_sse'].min()
        print(f"Current Generation: {current_gen}")
        print(f"Number of valid runs completed: {len(valid_runs_df)}")
        print(f"Best SSE found so far: {best_sse:.12f}")
        # Display the last few entries to see recent progress
        print("\nLast 5 entries in the ledger:")
        display(ledger_df.tail())
```

```
        else:  
            print("No valid runs completed yet in the ledger.")  
        else:  
            print(f"Ledger file not found at {LEDGER_FILE}. The hunt might not have started or completed any runs yet.")
```

```
Ledger file not found at /content/sweep_runs/SNCGL_ADAPTIVE_HUNT_012/ledger_SNCGL_ADAPTIVE_HUNT_012.csv. The hunt might not have started or completed any runs yet.
```