```
"""
LFT Analysis Code for Loophole-Free Bell Tests
=============================================
This code implements the analysis of loophole-free Bell test data
from Hensen et al. (2015) and Giustina et al. (2015) to validate
Logic Field Theory predictions.

Author: James D. Longmire, Jr.
Date: February 2025
"""

import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats
from scipy.optimize import minimize, curve_fit
import pandas as pd
from scipy.special import logsumexp
import os
import h5py

# Set high precision for calculations
np.set_printoptions(precision=16)

# Constants
SQRT2 = np.sqrt(2)
QM_CHSH = 2 * SQRT2  # Standard QM prediction for CHSH
COEF_THEORETICAL = 0.0415  # Theoretical coefficient from LFT

# Experimental data
# -----------------
# Hensen et al. (2015)
HENSEN_S = 2.42
HENSEN_ERROR = 0.20
HENSEN_FIDELITY = 0.92
HENSEN_TRIALS = 245

# Giustina et al. (2015)
GIUSTINA_S = 2.828
GIUSTINA_ERROR = 0.0005
GIUSTINA_TRIALS = 10**9
GIUSTINA_FIDELITY = 1.0

# Bell test angles (optimal settings)
THETA_1 = 0  # radians
THETA_2 = np.pi/4
PHI_1 = np.pi/8
PHI_2 = -np.pi/8


# ============================================================
# Core LFT calculation functions
# ============================================================
```

```
def qm_probability(a, b, theta, phi):
    """
    Calculate quantum mechanical probability for outcomes a,b given angles theta,phi
    for the Bell state (|01⟩ - |10⟩)/√2

    Parameters:
    -----------
    a, b : int (0 or 1)
        Measurement outcomes
    theta, phi : float
        Measurement angles in radians

    Returns:
    --------
    float
        Probability according to standard quantum mechanics
    """
    phase_adj = (a * np.pi/2) + (b * np.pi/2)
    return 0.5 * (np.sin((theta - phi)/2 + phase_adj))**2


def lft_probability(a, b, theta, phi, n_eff):
    """
    Calculate LFT probability with epsilon correction

    Parameters:
    -----------
    a, b : int (0 or 1)
        Measurement outcomes
    theta, phi : float
        Measurement angles in radians
    n_eff : float
        Effective dimensionality parameter

    Returns:
    --------
    float
        Probability according to LFT
    """
    p_qm = qm_probability(a, b, theta, phi)

    # LFT correction term
    epsilon = 1/n_eff
    correction = ((np.log(n_eff))**2 / n_eff) * p_qm * (1 - p_qm)

    # Return modified probability
    return p_qm + correction


def calculate_correlation(theta, phi, model="QM", n_eff=None, fidelity=1.0):
    """
```

```
    Calculate correlation E(θ,φ) for given angles using specified model

    Parameters:
    -----------
    theta, phi : float
        Measurement angles in radians
    model : str, "QM" or "LFT"
        Theoretical model to use
    n_eff : float, optional (required if model="LFT")
        Effective dimensionality parameter
    fidelity : float, default=1.0
        State fidelity factor

    Returns:
    --------
    float
        Correlation value E(θ,φ)
    """
    # Calculate all outcome probabilities
    if model == "QM":
        p00 = qm_probability(0, 0, theta, phi)
        p01 = qm_probability(0, 1, theta, phi)
        p10 = qm_probability(1, 0, theta, phi)
        p11 = qm_probability(1, 1, theta, phi)
    else:  # LFT
        if n_eff is None:
            raise ValueError("n_eff must be provided for LFT calculations")
        p00 = lft_probability(0, 0, theta, phi, n_eff)
        p01 = lft_probability(0, 1, theta, phi, n_eff)
        p10 = lft_probability(1, 0, theta, phi, n_eff)
        p11 = lft_probability(1, 1, theta, phi, n_eff)

    # Normalize probabilities to ensure they sum to 1
    total_p = p00 + p01 + p10 + p11
    p00 /= total_p
    p01 /= total_p
    p10 /= total_p
    p11 /= total_p

    # Calculate correlation and apply fidelity
    corr = p00 + p11 - p01 - p10
    return corr * fidelity


def calculate_chsh(model="QM", n_eff=None, fidelity=1.0):
    """
    Calculate CHSH value for given model and parameters

    Parameters:
    -----------
    model : str, "QM" or "LFT"
        Theoretical model to use
```

```
    n_eff : float, optional (required if model="LFT")
        Effective dimensionality parameter
    fidelity : float, default=1.0
        State fidelity factor

    Returns:
    --------
    float
        CHSH parameter S
    """
    e1 = calculate_correlation(THETA_1, PHI_1, model, n_eff, fidelity)
    e2 = calculate_correlation(THETA_1, PHI_2, model, n_eff, fidelity)
    e3 = calculate_correlation(THETA_2, PHI_1, model, n_eff, fidelity)
    e4 = calculate_correlation(THETA_2, PHI_2, model, n_eff, fidelity)

    return abs(e1 + e2 + e3 - e4)


def lft_chsh_analytical(n_eff, fidelity=1.0):
    """
    Calculate the analytical LFT prediction for CHSH using the formula:
    S_LFT ≈ 2√2 + 0.0415·(ln(n_eff))²/n_eff

    Parameters:
    -----------
    n_eff : float
        Effective dimensionality parameter
    fidelity : float, default=1.0
        State fidelity factor

    Returns:
    --------
    float
        CHSH parameter S according to the analytical formula
    """
    qm_value = 2 * SQRT2
    correction = COEF_THEORETICAL * ((np.log(n_eff))**2 / n_eff)
    return (qm_value + correction) * fidelity


# ==================================================================
# Statistical analysis functions
# ==================================================================

def log_likelihood(n_eff, s_exp, s_error, fidelity=1.0):
    """
    Log-likelihood function for Gaussian errors

    Parameters:
    -----------
    n_eff : float
        Effective dimensionality parameter
```

```
    s_exp : float
        Experimentally measured S value
    s_error : float
        Experimental error on S
    fidelity : float, default=1.0
        State fidelity factor

    Returns:
    --------
    float
        Log-likelihood value
    """
    s_lft = lft_chsh_analytical(n_eff, fidelity)
    return -0.5 * ((s_exp - s_lft) / s_error)**2


def find_best_fit_n_eff(s_exp, s_error, fidelity=1.0):
    """
    Find the best-fit n_eff value using maximum likelihood estimation

    Parameters:
    -----------
    s_exp : float
        Experimentally measured S value
    s_error : float
        Experimental error on S
    fidelity : float, default=1.0
        State fidelity factor

    Returns:
    --------
    tuple
        (best_n_eff, log_likelihood, confidence_interval)
    """
    # Objective function to minimize (negative log-likelihood)
    def objective(log_n):
        return -log_likelihood(np.exp(log_n), s_exp, s_error, fidelity)

    # Search in log space (more stable numerically)
    result = minimize(objective, np.log(1e5), method='BFGS')
    best_log_n = result.x[0]
    best_n_eff = np.exp(best_log_n)
    best_ll = -result.fun

    # Find confidence interval using likelihood ratio test
    # We use the fact that -2(L(n) - L(n_best)) ~ χ²(1)
    chi2_critical = 3.84  # 95% confidence for 1 degree of freedom
    min_ll = best_ll - chi2_critical/2

    # Grid search for confidence interval bounds
    log_n_grid = np.linspace(best_log_n - 3, best_log_n + 3, 1000)
    ll_values = [-objective(log_n) for log_n in log_n_grid]
```

```
    # Find where log-likelihood crosses the threshold
    valid_indices = np.where(np.array(ll_values) >= min_ll)[0]
    lower_idx = valid_indices[0]
    upper_idx = valid_indices[-1]
    lower_bound = np.exp(log_n_grid[lower_idx])
    upper_bound = np.exp(log_n_grid[upper_idx])

    confidence_interval = (lower_bound, upper_bound)

    return best_n_eff, best_ll, confidence_interval


def z_score(s_exp, s_theory, s_error):
    """
    Calculate the z-score for a theoretical prediction

    Parameters:
    -----------
    s_exp : float
        Experimentally measured S value
    s_theory : float
        Theoretical prediction for S
    s_error : float
        Experimental error on S

    Returns:
    --------
    float
        Z-score value
    """
    return (s_exp - s_theory) / s_error


def power_analysis(delta_s, alpha=0.05, power=0.95):
    """
    Calculate the required sample size for detecting a deviation

    Parameters:
    -----------
    delta_s : float
        Expected deviation in S value
    alpha : float, default=0.05
        Significance level
    power : float, default=0.95
        Desired statistical power

    Returns:
    --------
    float
        Required number of trials
    """
```

```python
    z_alpha = stats.norm.ppf(1 - alpha/2)  # Two-tailed test
    z_power = stats.norm.ppf(power)

    # For a given sample size N, standard error scales as 1/sqrt(N)
    # We need delta_s/sigma > z_alpha + z_power
    # This gives N > ((z_alpha + z_power)/delta_s)^2

    required_n = ((z_alpha + z_power)/delta_s)**2
    return required_n


def extract_coefficient(n_eff, delta_s):
    """
    Extract the coefficient from the (ln n)²/n scaling

    Parameters:
    -----------
    n_eff : float
        Effective dimensionality parameter
    delta_s : float
        Deviation in S value

    Returns:
    --------
    float
        Extracted coefficient
    """
    return delta_s * n_eff / (np.log(n_eff)**2)


# ===============================================================
# Main analysis functions
# ===============================================================

def analyze_hensen_experiment():
    """
    Analyze the Hensen et al. (2015) experiment

    Returns:
    --------
    dict
        Analysis results
    """
    print("Analyzing Hensen et al. (2015) experiment...")

    # QM prediction with fidelity correction
    qm_pred = QM_CHSH * HENSEN_FIDELITY
    print(f"QM prediction (with fidelity={HENSEN_FIDELITY}): {qm_pred:.6f}")

    # Calculate z-score for QM
    qm_z = z_score(HENSEN_S, qm_pred, HENSEN_ERROR)
    print(f"QM z-score: {qm_z:.4f}σ")
```

```
    # Test different n_eff values
    n_eff_values = [1e3, 1e4, 1e5, 1e6, 1e7]
    results = []

    for n in n_eff_values:
        # Calculate LFT prediction
        lft_pred = lft_chsh_analytical(n, HENSEN_FIDELITY)

        # Calculate z-score
        lft_z = z_score(HENSEN_S, lft_pred, HENSEN_ERROR)

        print(f"n_eff = 10^{np.log10(n):.0f}: S = {lft_pred:.6f}, z-score = {lft_z:.4f}σ")

        results.append({
            'n_eff': n,
            'S_pred': lft_pred,
            'z_score': lft_z,
            'p_value': 2 * stats.norm.sf(abs(lft_z))  # Two-tailed p-value
        })

    # Find best-fit n_eff
    best_n, best_ll, ci = find_best_fit_n_eff(HENSEN_S, HENSEN_ERROR, HENSEN_FIDELITY)
    print(f"Best-fit n_eff: {best_n:.2e}, 95% CI: [{ci[0]:.2e}, {ci[1]:.2e}]")

    return {
        'experiment': 'Hensen et al. (2015)',
        'S_exp': HENSEN_S,
        'S_error': HENSEN_ERROR,
        'fidelity': HENSEN_FIDELITY,
        'QM_prediction': qm_pred,
        'QM_z_score': qm_z,
        'LFT_results': results,
        'best_fit_n_eff': best_n,
        'confidence_interval': ci
    }


def analyze_giustina_experiment():
    """
    Analyze the Giustina et al. (2015) experiment

    Returns:
    --------
    dict
        Analysis results
    """
    print("\nAnalyzing Giustina et al. (2015) experiment...")

    # QM prediction (no fidelity correction needed)
    qm_pred = QM_CHSH
    print(f"QM prediction: {qm_pred:.6f}")
```

```
# Calculate z-score for QM
qm_z = z_score(GIUSTINA_S, qm_pred, GIUSTINA_ERROR)
print(f"QM z-score: {qm_z:.4f}σ")

# Test different n_eff values
n_eff_values = [1e3, 1e4, 1e5, 1e6, 1e7]
results = []

for n in n_eff_values:
    # Calculate LFT prediction
    lft_pred = lft_chsh_analytical(n)

    # Calculate z-score
    lft_z = z_score(GIUSTINA_S, lft_pred, GIUSTINA_ERROR)

    # Calculate p-value
    p_val = 2 * stats.norm.sf(abs(lft_z))  # Two-tailed p-value

    print(f"n_eff = 10^{np.log10(n):.0f}: S = {lft_pred:.6f}, z-score = {lft_z:.4f}σ, p = {p_val:.4e}")

    results.append({
        'n_eff': n,
        'S_pred': lft_pred,
        'z_score': lft_z,
        'p_value': p_val
    })

# Find best-fit n_eff
best_n, best_ll, ci = find_best_fit_n_eff(GIUSTINA_S, GIUSTINA_ERROR)
print(f"Best-fit n_eff: {best_n:.2e}, 95% CI: [{ci[0]:.2e}, {ci[1]:.2e}]")

# For n_eff = 10^4, check if we can exclude it
n4_pred = lft_chsh_analytical(1e4)
n4_z = z_score(GIUSTINA_S, n4_pred, GIUSTINA_ERROR)
p_val_n4 = 2 * stats.norm.sf(abs(n4_z))

print(f"Test of n_eff ≤ 10^4:")
print(f"  z-score: {n4_z:.4f}σ, p-value: {p_val_n4:.4e}")
if abs(n4_z) > 4:
    print(f"  Excluded at >4σ confidence")

return {
    'experiment': 'Giustina et al. (2015)',
    'S_exp': GIUSTINA_S,
    'S_error': GIUSTINA_ERROR,
    'fidelity': GIUSTINA_FIDELITY,
    'QM_prediction': qm_pred,
    'QM_z_score': qm_z,
    'LFT_results': results,
    'best_fit_n_eff': best_n,
    'confidence_interval': ci,
```

```
        'n4_exclusion_z': n4_z,
        'n4_exclusion_p': p_val_n4
    }


def calculate_power_requirements():
    """
    Calculate required trial numbers for future experiments

    Returns:
    --------
    dict
        Power analysis results
    """
    print("\nPower analysis for future experiments...")

    # Calculate deviation for n_eff = 10^5
    delta_s = lft_chsh_analytical(1e5) - QM_CHSH
    print(f"Expected ΔS for n_eff = 10^5: {delta_s:.8e}")

    # Trial numbers to test
    trial_numbers = [1e9, 1e10, 1e11, 1e12]
    power_results = []

    for trials in trial_numbers:
        # Statistical error scales as 1/sqrt(N)
        sigma = 1/np.sqrt(trials)
        sigma_level = delta_s/sigma

        # Calculate power for 5σ detection
        power_5sigma = stats.norm.cdf(sigma_level - 5)

        # Calculate power for 95% significance
        power_95 = stats.norm.cdf(sigma_level - 1.96)

        print(f"{trials:.1e} trials: σ = {sigma:.8f}, detection at {sigma_level:.1f}σ, power(5σ) = {power_5sigma:.4f},
power(95%) = {power_95:.4f}")

        power_results.append({
            'trials': trials,
            'sigma': sigma,
            'sigma_level': sigma_level,
            'power_5sigma': power_5sigma,
            'power_95': power_95
        })

    # Calculate required trials for 5σ detection with 99% power
    required_trials = power_analysis(delta_s, alpha=5.7e-7, power=0.99)  # 5σ corresponds to p=5.7e-7
    print(f"Required trials for 5σ detection with 99% power: {required_trials:.2e}")

    return {
        'delta_s': delta_s,
```

```
        'power_results': power_results,
        'required_trials': required_trials
    }


def coefficient_analysis():
    """
    Analyze the coefficient in the (ln n)²/n scaling

    Returns:
    --------
    dict
        Coefficient analysis results
    """
    print("\nCoefficient analysis...")

    n_eff_values = [1e3, 1e4, 1e5, 1e6, 1e7]
    coefficients = []

    for n in n_eff_values:
        # Calculate S value and deviation
        lft_s = lft_chsh_analytical(n)
        delta_s = lft_s - QM_CHSH

        # Extract coefficient
        coef = extract_coefficient(n, delta_s)

        coefficients.append(coef)
        print(f"n_eff = 10^{np.log10(n):.0f}: coefficient = {coef:.6f}")

    avg_coef = np.mean(coefficients)
    print(f"Average coefficient: {avg_coef:.6f}")
    print(f"Expected from theory: {COEF_THEORETICAL}")
    print(f"Difference: {avg_coef - COEF_THEORETICAL:.6f} ({(avg_coef -
COEF_THEORETICAL)/COEF_THEORETICAL*100:.2f}%)")

    return {
        'n_eff_values': n_eff_values,
        'coefficients': coefficients,
        'average': avg_coef,
        'theoretical': COEF_THEORETICAL
    }


def compare_with_paper_prediction():
    """
    Compare our calculations with the paper's prediction

    Returns:
    --------
    dict
        Comparison results
```

```
    """
    print("\nComparison with paper prediction for n_eff = 10^5:")

    n_paper = 1e5
    s_paper = 2.8288  # From LFT paper
    s_our = lft_chsh_analytical(n_paper)

    print(f"Paper prediction: S = {s_paper}")
    print(f"Our calculation: S = {s_our:.8f}")

    diff_abs = s_our - s_paper
    diff_rel = diff_abs / s_paper * 100
    print(f"Difference: {diff_abs:.8e} ({diff_rel:.8f}%)")

    # Calculate what coefficient would be needed for the paper's value
    delta_paper = s_paper - QM_CHSH
    delta_our = s_our - QM_CHSH

    coef_needed = extract_coefficient(n_paper, delta_paper)
    print(f"Required coefficient for paper's value: {coef_needed:.6f}")

    return {
        'paper_value': s_paper,
        'our_value': s_our,
        'difference': diff_abs,
        'relative_difference': diff_rel,
        'required_coefficient': coef_needed
    }


def simple_polarization_analysis():
    """
    Analyze simpler polarization measurement experiment

    Returns:
    --------
    dict
        Polarization analysis results
    """
    print("\nPolarization measurement analysis...")

    # For a superposition state |H⟩ + |V⟩/√2, QM predicts P(H) = 0.5
    # In LFT, this becomes P(H) ≈ 0.5 + (ln n)²/(4n)

    n_eff_values = [1e3, 1e4, 1e5, 1e6]
    p_h_values = []

    for n in n_eff_values:
        # Calculate probability correction
        correction = ((np.log(n))**2) / (4*n)
        p_h = 0.5 + correction
```

```
        # Calculate required trials for 5σ detection
        required_trials = power_analysis(correction, alpha=5.7e-7, power=0.95)

        p_h_values.append({
            'n_eff': n,
            'P(H)': p_h,
            'delta_P': correction,
            'required_trials': required_trials
        })

        print(f"n_eff = 10^{np.log10(n):.0f}: P(H) = {p_h:.8f}, ΔP = {correction:.8e}")
        print(f"  Required trials for 5σ detection: {required_trials:.2e}")

    return {
        'p_h_values': p_h_values
    }


def create_visualizations(results_hensen, results_giustina, power_results, coef_results):
    """
    Create visualizations for the analysis

    Parameters:
    -----------
    results_hensen : dict
        Results from Hensen experiment analysis
    results_giustina : dict
        Results from Giustina experiment analysis
    power_results : dict
        Results from power analysis
    coef_results : dict
        Results from coefficient analysis
    """
    print("\nCreating visualizations...")

    # Figure 1: CHSH values for different n_eff
    plt.figure(figsize=(10, 6))

    # Create x-axis for plotting
    log_n = np.log10(np.logspace(3, 7, 1000))
    n_values = 10**log_n

    # Calculate predictions for different n_eff values
    s_values = [lft_chsh_analytical(n) for n in n_values]

    # Plot LFT model curve
    plt.plot(log_n, s_values, 'b-', label='LFT Prediction')

    # Plot standard QM prediction
    plt.axhline(y=QM_CHSH, color='r', linestyle='--',
               label=f'QM Prediction: {QM_CHSH:.6f}')
```

```python
# Plot Giustina experimental result with error bars
plt.errorbar([3, 7], [GIUSTINA_S, GIUSTINA_S],
            yerr=[GIUSTINA_ERROR, GIUSTINA_ERROR],
            fmt='go', label=f'Giustina et al.: {GIUSTINA_S} ± {GIUSTINA_ERROR}')

# Add rejected region for n_eff ≤ 10^4
plt.axvspan(3, 4, alpha=0.2, color='red', label='Excluded: n_eff ≤ 10^4')

# Add best-fit and confidence interval
plt.axvline(x=np.log10(results_giustina['best_fit_n_eff']), color='g', linestyle='-',
           label=f'Best fit: n_eff ≈ {results_giustina["best_fit_n_eff"]:.2e}')

ci = results_giustina['confidence_interval']
plt.axvspan(np.log10(ci[0]), np.log10(ci[1]), alpha=0.3, color='green',
           label=f'95% CI: [{ci[0]:.2e}, {ci[1]:.2e}]')

plt.xlabel('log₁₀(n_eff)')
plt.ylabel('CHSH Bell Parameter (S)')
plt.title('Logic Field Theory Predictions vs. Experimental Results')
plt.grid(True)
plt.legend(loc='upper center', bbox_to_anchor=(0.5, -0.15), ncol=2)
plt.tight_layout()

plt.savefig('LFT_bell_test_validation.png', dpi=300, bbox_inches='tight')

# Figure 2: Statistical power analysis
plt.figure(figsize=(8, 5))

trial_numbers = [r['trials'] for r in power_results['power_results']]
sigma_levels = [r['sigma_level'] for r in power_results['power_results']]
power_values = [r['power_95'] for r in power_results['power_results']]

plt.semilogx(trial_numbers, sigma_levels, 'b-o', label='Detection significance')
plt.axhline(y=5, color='r', linestyle='--', label='5σ threshold')

plt.xlabel('Number of trials')
plt.ylabel('Detection significance (σ)')
plt.title('Statistical Power for Detecting LFT Effects (n_eff = 10^5)')
plt.grid(True)
plt.legend()
plt.tight_layout()

plt.savefig('LFT_statistical_power.png', dpi=300)

# Figure 3: Coefficient analysis
plt.figure(figsize=(8, 5))

n_values = coef_results['n_eff_values']
coefficients = coef_results['coefficients']

plt.semilogx(n_values, coefficients, 'bo-', label='Extracted coefficients')
plt.axhline(y=COEF_THEORETICAL, color='r', linestyle='--',
```

```python
            label=f'Theoretical value: {COEF_THEORETICAL}')

    plt.xlabel('n_eff')
    plt.ylabel('Coefficient value')
    plt.title('Coefficient in ΔS = c·(ln n)²/n Scaling')
    plt.grid(True)
    plt.legend()
    plt.tight_layout()

    plt.savefig('LFT_coefficient_analysis.png', dpi=300)

    print("Visualizations saved to disk.")


def save_results_to_file(results, filename='lft_analysis_results.json'):
    """
    Save analysis results to a JSON file

    Parameters:
    -----------
    results : dict
        Analysis results to save
    filename : str, default='lft_analysis_results.json'
        Output filename
    """
    import json

    # Convert numpy values to Python native types
    def numpy_to_python(obj):
        if isinstance(obj, np.ndarray):
            return obj.tolist()
        elif isinstance(obj, np.integer):
            return int(obj)
        elif isinstance(obj, np.floating):
            return float(obj)
        elif isinstance(obj, dict):
            return {k: numpy_to_python(v) for k, v in obj.items()}
        elif isinstance(obj, list):
            return [numpy_to_python(item) for item in obj]
        else:
            return obj

    # Convert results to JSON-serializable format
    serializable_results = numpy_to_python(results)

    # Save to file
    with open(filename, 'w') as f:
        json.dump(serializable_results, f, indent=2)

    print(f"Results saved to {filename}")
```

```python
def load_experimental_data(data_dir='./data'):
    """
    Load experimental data from original sources if available

    Parameters:
    -----------
    data_dir : str, default='./data'
        Directory containing experimental data files

    Returns:
    --------
    dict
        Loaded experimental data
    """
    data = {
        'hensen': None,
        'giustina': None
    }

    # Paths to data files (if available)
    hensen_file = os.path.join(data_dir, 'hensen_2015_data.csv')
    giustina_file = os.path.join(data_dir, 'giustina_2015_data.h5')

    # Check if files exist and load data
    if os.path.exists(hensen_file):
        try:
            # The format here depends on how the Hensen data is stored
            # This is a placeholder - adjust based on actual data format
            hensen_data = pd.read_csv(hensen_file)
            data['hensen'] = hensen_data
            print(f"Loaded Hensen et al. data from {hensen_file}")
        except Exception as e:
            print(f"Error loading Hensen data: {e}")
    else:
        print(f"Hensen data file not found: {hensen_file}")
        print("Using published summary statistics instead.")

    if os.path.exists(giustina_file):
        try:
            # The format here depends on how the Giustina data is stored
            # This is a placeholder - adjust based on actual data format
            with h5py.File(giustina_file, 'r') as f:
                giustina_data = {key: f[key][()] for key in f.keys()}
            data['giustina'] = giustina_data
            print(f"Loaded Giustina et al. data from {giustina_file}")
        except Exception as e:
            print(f"Error loading Giustina data: {e}")
    else:
        print(f"Giustina data file not found: {giustina_file}")
        print("Using published summary statistics instead.")

    return data
```

```
def process_raw_hensen_data(data):
    """
    Process raw Hensen et al. experimental data

    This function would implement the exact data processing steps used by Hensen et al.
    to calculate their CHSH value from the raw experimental data.

    Parameters:
    -----------
    data : DataFrame
        Raw experimental data

    Returns:
    --------
    dict
        Processed results including CHSH value
    """
    # Note: This is a placeholder implementation
    # In a real analysis, this would implement the exact data processing
    # procedure described in the Hensen et al. paper

    # Example processing steps:
    # 1. Filter valid events
    # 2. Group by measurement settings
    # 3. Calculate correlations for each setting pair
    # 4. Combine into CHSH parameter

    # For now, we just return the published values
    return {
        'S': HENSEN_S,
        'error': HENSEN_ERROR,
        'fidelity': HENSEN_FIDELITY,
        'raw_data_available': data is not None
    }


def process_raw_giustina_data(data):
    """
    Process raw Giustina et al. experimental data

    This function would implement the exact data processing steps used by Giustina et al.
    to calculate their CHSH value from the raw experimental data.

    Parameters:
    -----------
    data : dict
        Raw experimental data

    Returns:
    --------
```

```
    dict
        Processed results including CHSH value
    """
    # Note: This is a placeholder implementation
    # In a real analysis, this would implement the exact data processing
    # procedure described in the Giustina et al. paper

    # Example processing steps:
    # 1. Extract coincidence counts for each measurement setting
    # 2. Calculate correlations for each setting pair
    # 3. Combine into CHSH parameter
    # 4. Calculate statistical error

    # For now, we just return the published values
    return {
        'S': GIUSTINA_S,
        'error': GIUSTINA_ERROR,
        'fidelity': GIUSTINA_FIDELITY,
        'raw_data_available': data is not None
    }


def monte_carlo_simulation(n_eff, num_trials=10**6, fidelity=1.0):
    """
    Perform Monte Carlo simulation of a Bell test with LFT corrections

    Parameters:
    -----------
    n_eff : float
        Effective dimensionality parameter
    num_trials : int, default=10^6
        Number of simulated measurements
    fidelity : float, default=1.0
        State fidelity factor

    Returns:
    --------
    dict
        Simulation results
    """
    print(f"\nPerforming Monte Carlo simulation with n_eff={n_eff}, trials={num_trials}...")

    # CHSH measurement settings
    angle_pairs = [
        (THETA_1, PHI_1),  # Setting pair 1
        (THETA_1, PHI_2),  # Setting pair 2
        (THETA_2, PHI_1),  # Setting pair 3
        (THETA_2, PHI_2)   # Setting pair 4
    ]

    # Initialize results
    correlations_qm = []
```

```
correlations_lft = []

# For each angle pair
for i, (theta, phi) in enumerate(angle_pairs):
    print(f"  Simulating angle pair {i+1}: (θ={theta:.4f}, φ={phi:.4f})...")

    # Arrays to store measurement outcomes
    outcomes_qm = np.zeros((num_trials, 2), dtype=int)
    outcomes_lft = np.zeros((num_trials, 2), dtype=int)

    # Generate random numbers for the trials
    random_values = np.random.random(num_trials)

    # Calculate outcome probabilities
    p00_qm = qm_probability(0, 0, theta, phi) * fidelity
    p01_qm = qm_probability(0, 1, theta, phi) * fidelity
    p10_qm = qm_probability(1, 0, theta, phi) * fidelity
    p11_qm = qm_probability(1, 1, theta, phi) * fidelity

    p00_lft = lft_probability(0, 0, theta, phi, n_eff) * fidelity
    p01_lft = lft_probability(0, 1, theta, phi, n_eff) * fidelity
    p10_lft = lft_probability(1, 0, theta, phi, n_eff) * fidelity
    p11_lft = lft_probability(1, 1, theta, phi, n_eff) * fidelity

    # Normalize probabilities
    sum_qm = p00_qm + p01_qm + p10_qm + p11_qm
    p00_qm /= sum_qm
    p01_qm /= sum_qm
    p10_qm /= sum_qm
    p11_qm /= sum_qm

    sum_lft = p00_lft + p01_lft + p10_lft + p11_lft
    p00_lft /= sum_lft
    p01_lft /= sum_lft
    p10_lft /= sum_lft
    p11_lft /= sum_lft

    # Determine QM outcomes based on random values
    for j in range(num_trials):
        r = random_values[j]
        if r < p00_qm:
            outcomes_qm[j] = [0, 0]
        elif r < p00_qm + p01_qm:
            outcomes_qm[j] = [0, 1]
        elif r < p00_qm + p01_qm + p10_qm:
            outcomes_qm[j] = [1, 0]
        else:
            outcomes_qm[j] = [1, 1]

    # Determine LFT outcomes based on random values
    for j in range(num_trials):
        r = random_values[j]
```

```
            if r < p00_lft:
                outcomes_lft[j] = [0, 0]
            elif r < p00_lft + p01_lft:
                outcomes_lft[j] = [0, 1]
            elif r < p00_lft + p01_lft + p10_lft:
                outcomes_lft[j] = [1, 0]
            else:
                outcomes_lft[j] = [1, 1]

        # Calculate correlations from outcomes
        n00_qm = np.sum((outcomes_qm[:, 0] == 0) & (outcomes_qm[:, 1] == 0))
        n01_qm = np.sum((outcomes_qm[:, 0] == 0) & (outcomes_qm[:, 1] == 1))
        n10_qm = np.sum((outcomes_qm[:, 0] == 1) & (outcomes_qm[:, 1] == 0))
        n11_qm = np.sum((outcomes_qm[:, 0] == 1) & (outcomes_qm[:, 1] == 1))

        n00_lft = np.sum((outcomes_lft[:, 0] == 0) & (outcomes_lft[:, 1] == 0))
        n01_lft = np.sum((outcomes_lft[:, 0] == 0) & (outcomes_lft[:, 1] == 1))
        n10_lft = np.sum((outcomes_lft[:, 0] == 1) & (outcomes_lft[:, 1] == 0))
        n11_lft = np.sum((outcomes_lft[:, 0] == 1) & (outcomes_lft[:, 1] == 1))

        # Compute correlations E(θ,φ)
        corr_qm = (n00_qm + n11_qm - n01_qm - n10_qm) / num_trials
        corr_lft = (n00_lft + n11_lft - n01_lft - n10_lft) / num_trials

        correlations_qm.append(corr_qm)
        correlations_lft.append(corr_lft)

        print(f"    QM correlation: {corr_qm:.6f}")
        print(f"    LFT correlation: {corr_lft:.6f}")
        print(f"    Difference: {corr_lft - corr_qm:.8f}")

    # Calculate CHSH parameters
    S_qm = abs(correlations_qm[0] + correlations_qm[1] + correlations_qm[2] - correlations_qm[3])
    S_lft = abs(correlations_lft[0] + correlations_lft[1] + correlations_lft[2] - correlations_lft[3])

    print(f"Monte Carlo results:")
    print(f"  QM CHSH: {S_qm:.6f}")
    print(f"  LFT CHSH: {S_lft:.6f}")
    print(f"  Difference: {S_lft - S_qm:.8f}")

    # Compare with analytical predictions
    S_qm_analytical = QM_CHSH * fidelity
    S_lft_analytical = lft_chsh_analytical(n_eff, fidelity)

    print(f"Analytical predictions:")
    print(f"  QM CHSH: {S_qm_analytical:.6f}")
    print(f"  LFT CHSH: {S_lft_analytical:.6f}")
    print(f"  Difference: {S_lft_analytical - S_qm_analytical:.8f}")

    return {
        'n_eff': n_eff,
        'num_trials': num_trials,
```

```python
        'fidelity': fidelity,
        'correlations_qm': correlations_qm,
        'correlations_lft': correlations_lft,
        'S_qm_monte_carlo': S_qm,
        'S_lft_monte_carlo': S_lft,
        'delta_S_monte_carlo': S_lft - S_qm,
        'S_qm_analytical': S_qm_analytical,
        'S_lft_analytical': S_lft_analytical,
        'delta_S_analytical': S_lft_analytical - S_qm_analytical
    }


def run_full_analysis():
    """Run the complete analysis pipeline"""
    print("=" * 80)
    print("Logic Field Theory - Loophole-Free Bell Test Analysis")
    print("=" * 80)

    # Attempt to load raw experimental data
    exp_data = load_experimental_data()

    # Process raw data if available
    if exp_data['hensen'] is not None:
        hensen_processed = process_raw_hensen_data(exp_data['hensen'])
    else:
        hensen_processed = process_raw_hensen_data(None)

    if exp_data['giustina'] is not None:
        giustina_processed = process_raw_giustina_data(exp_data['giustina'])
    else:
        giustina_processed = process_raw_giustina_data(None)

    # Run individual analyses
    results_hensen = analyze_hensen_experiment()
    results_giustina = analyze_giustina_experiment()
    power_results = calculate_power_requirements()
    coef_results = coefficient_analysis()
    paper_comparison = compare_with_paper_prediction()
    polarization_results = simple_polarization_analysis()

    # Run Monte Carlo simulation for validation
    mc_results = monte_carlo_simulation(n_eff=1e5, num_trials=1e6)

    # Create visualizations
    create_visualizations(results_hensen, results_giustina, power_results, coef_results)

    # Combine all results
    all_results = {
        'hensen': results_hensen,
        'giustina': results_giustina,
        'power_analysis': power_results,
        'coefficient_analysis': coef_results,
```

```python
        'paper_comparison': paper_comparison,
        'polarization_analysis': polarization_results,
        'monte_carlo': mc_results,
        'raw_data_processing': {
            'hensen': hensen_processed,
            'giustina': giustina_processed
        }
    }

    # Save results to file
    save_results_to_file(all_results)

    print("\nAnalysis complete!")
    return all_results


if __name__ == "__main__":
    results = run_full_analysis()
```