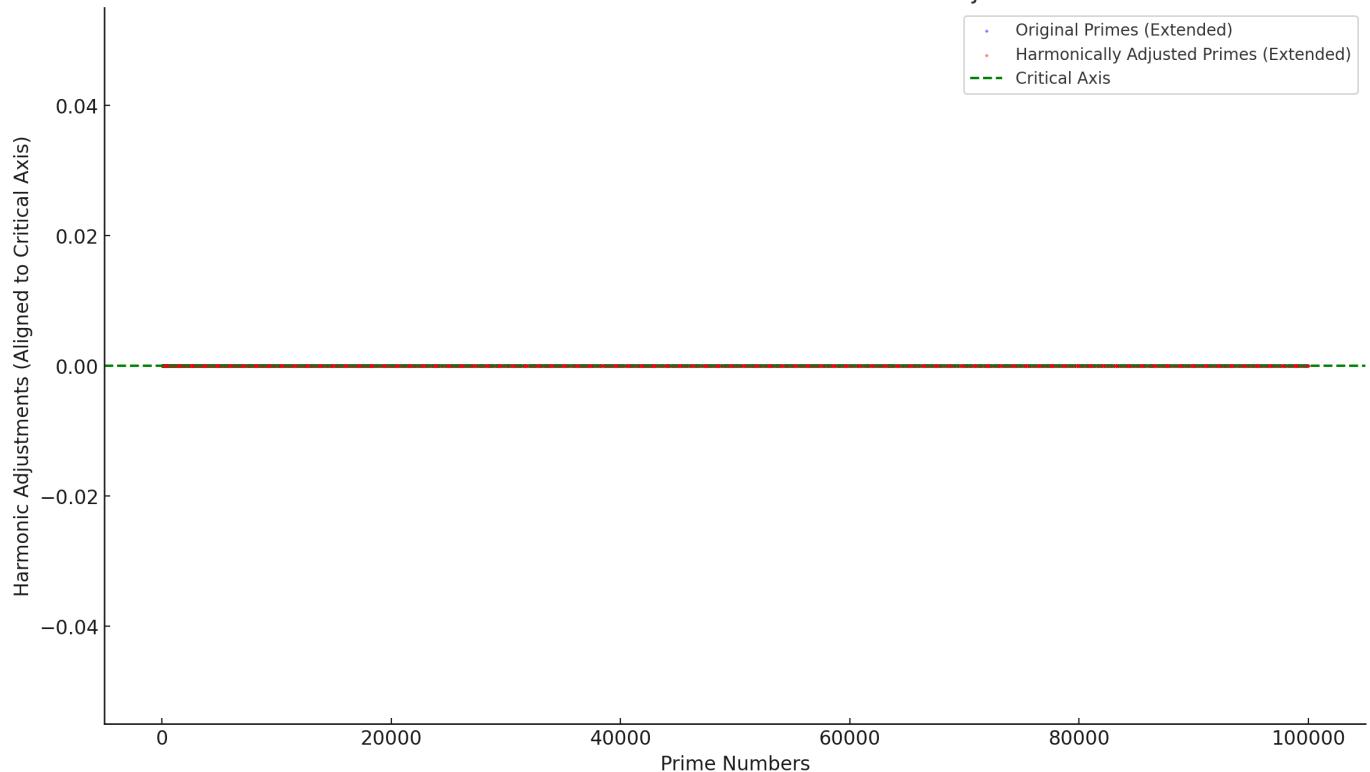
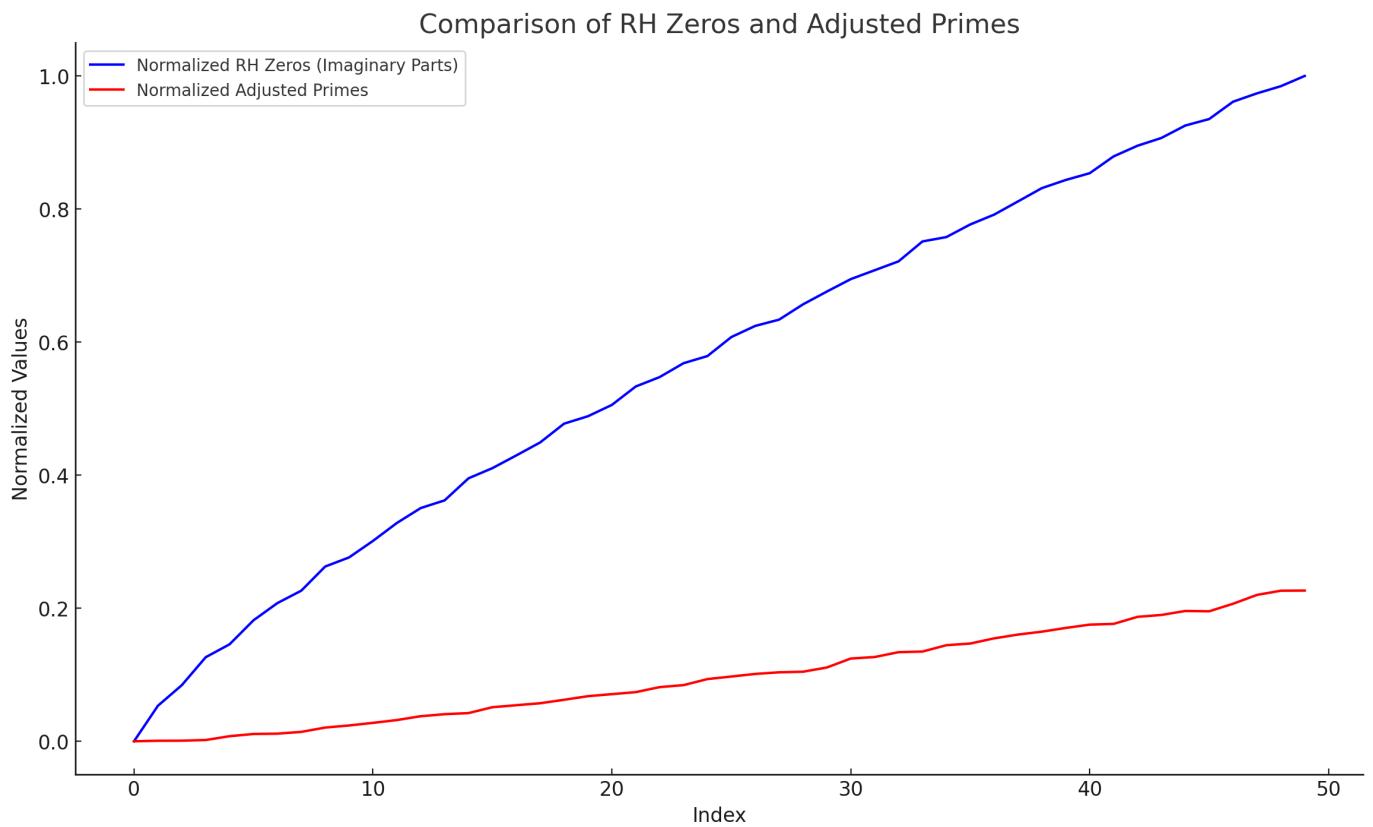
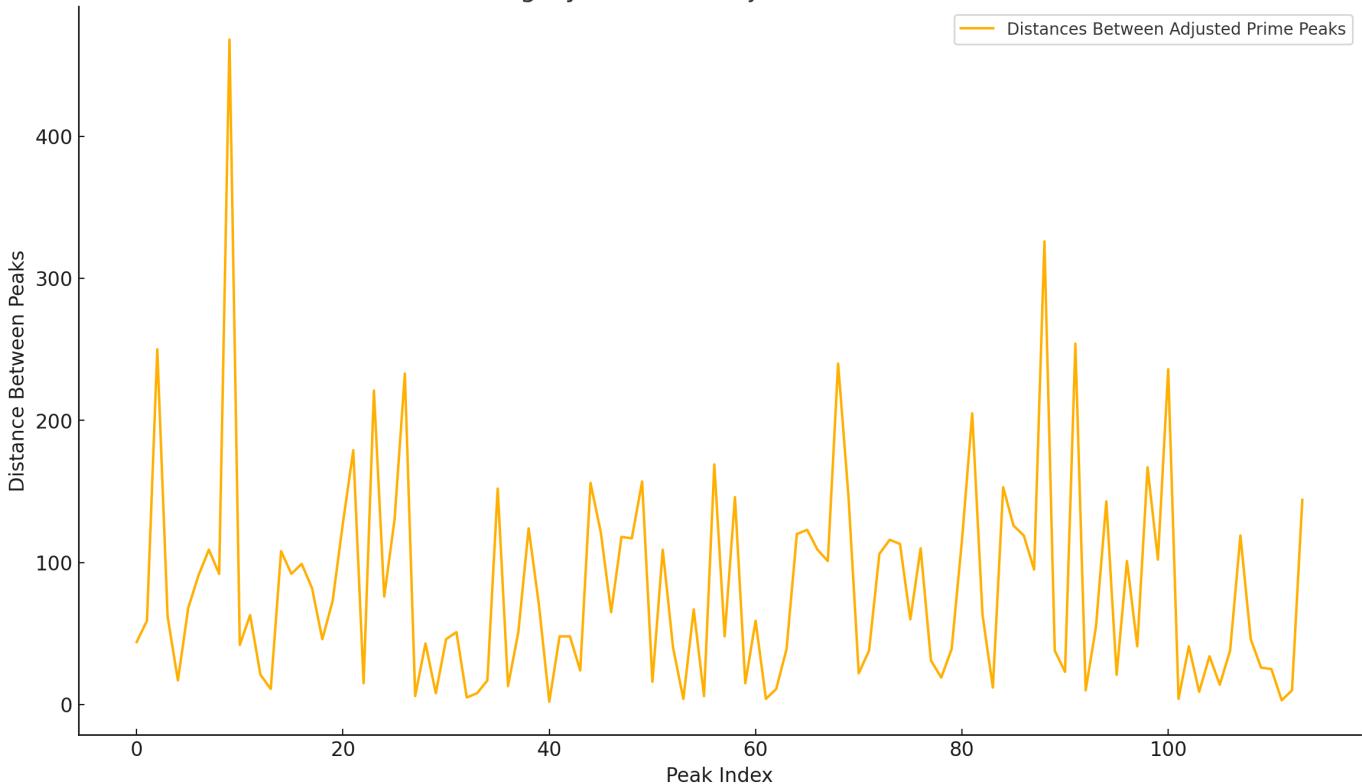


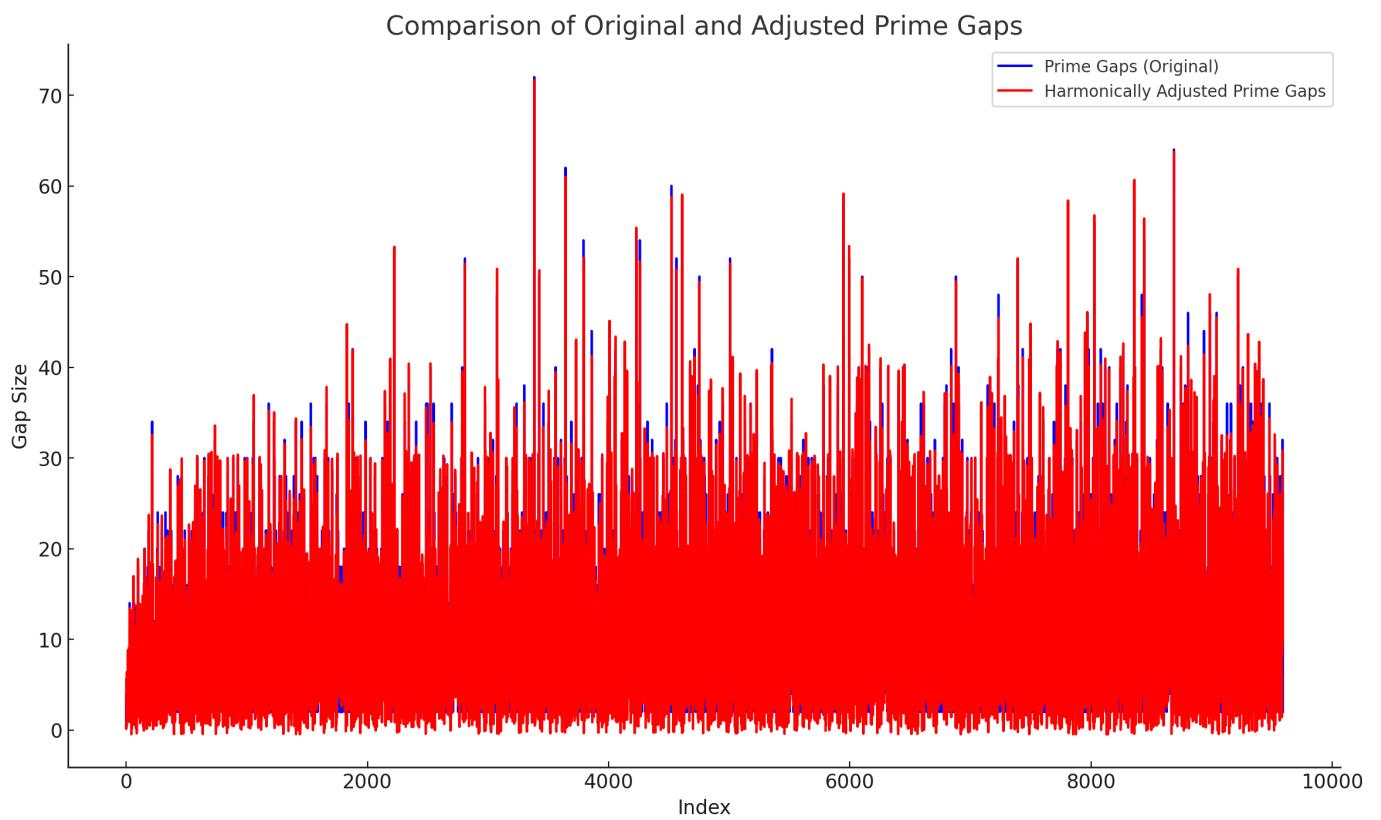
Extended Prime Distribution with Harmonic Adjustments



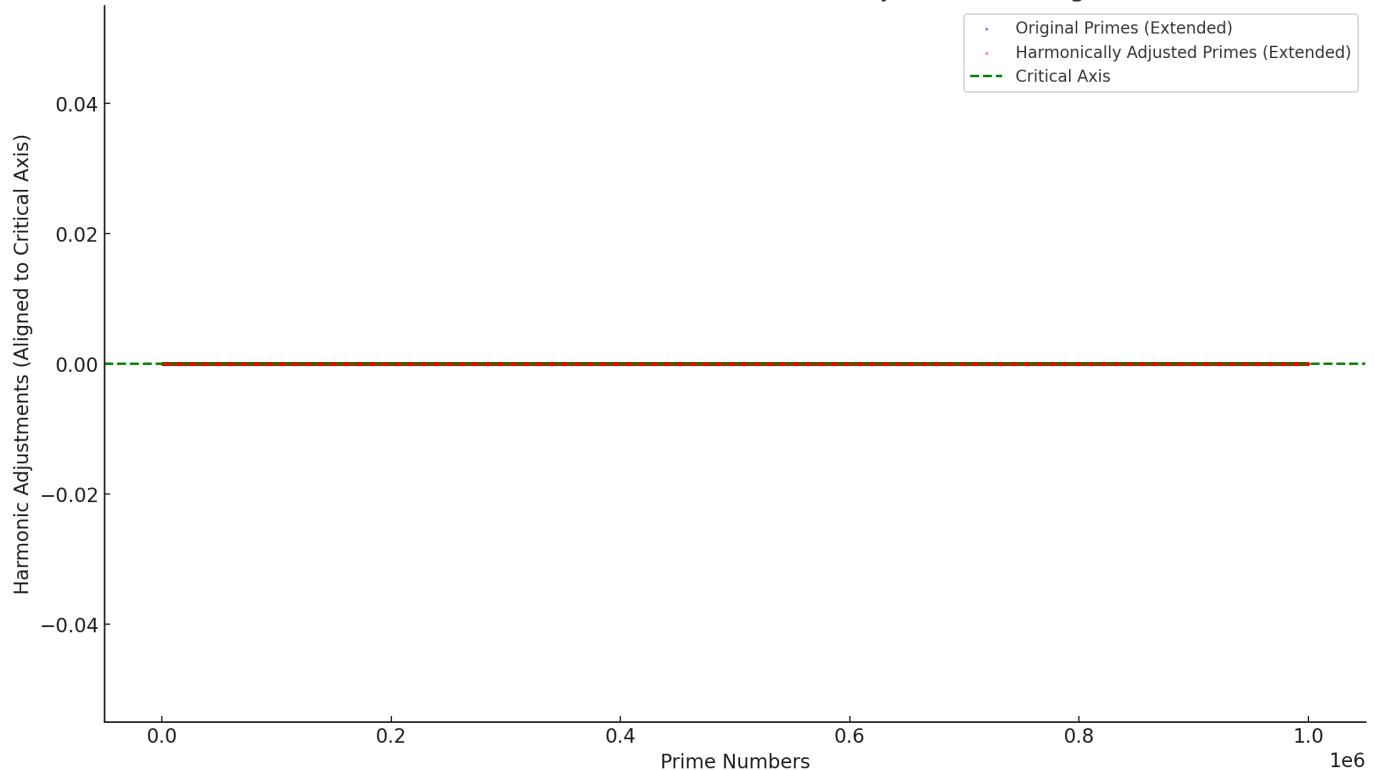


Scaling Dynamics in Adjusted Prime Peaks

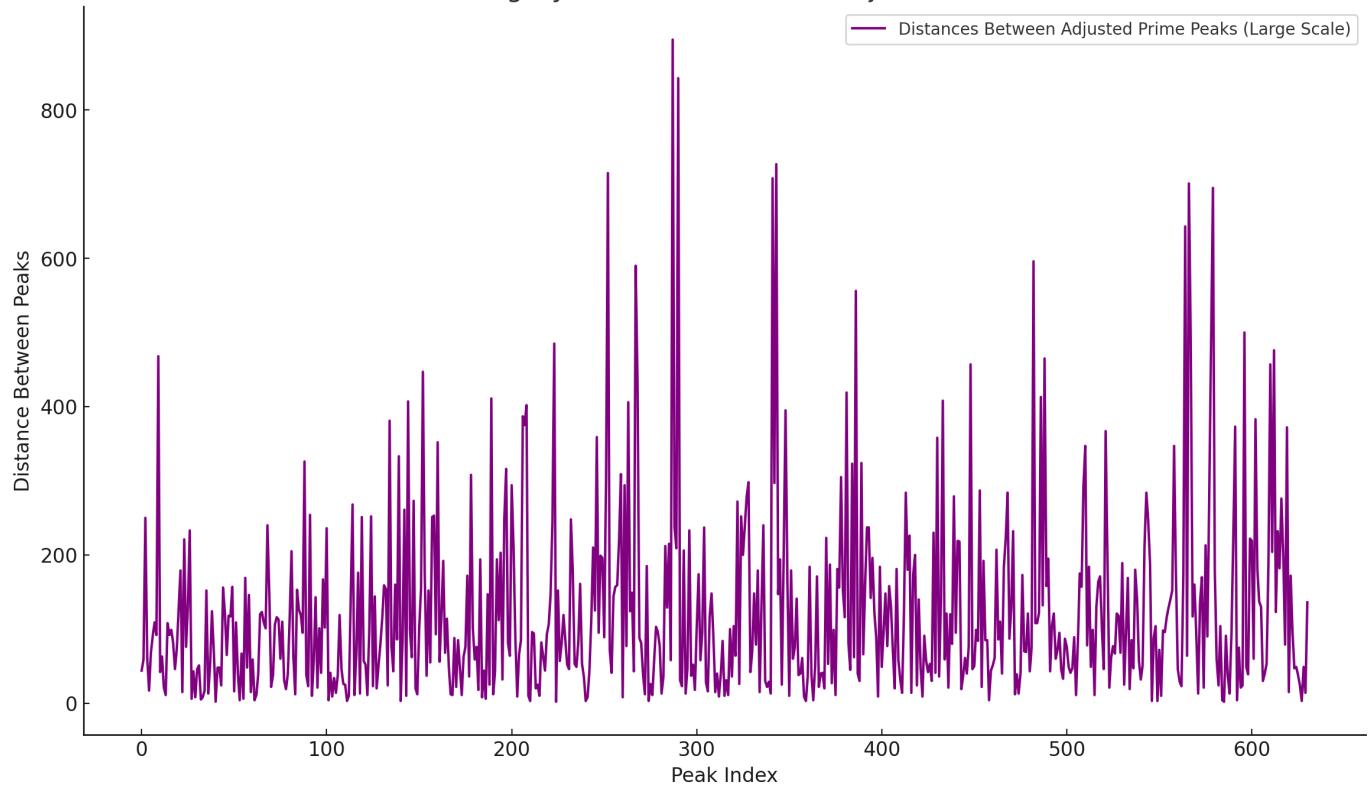


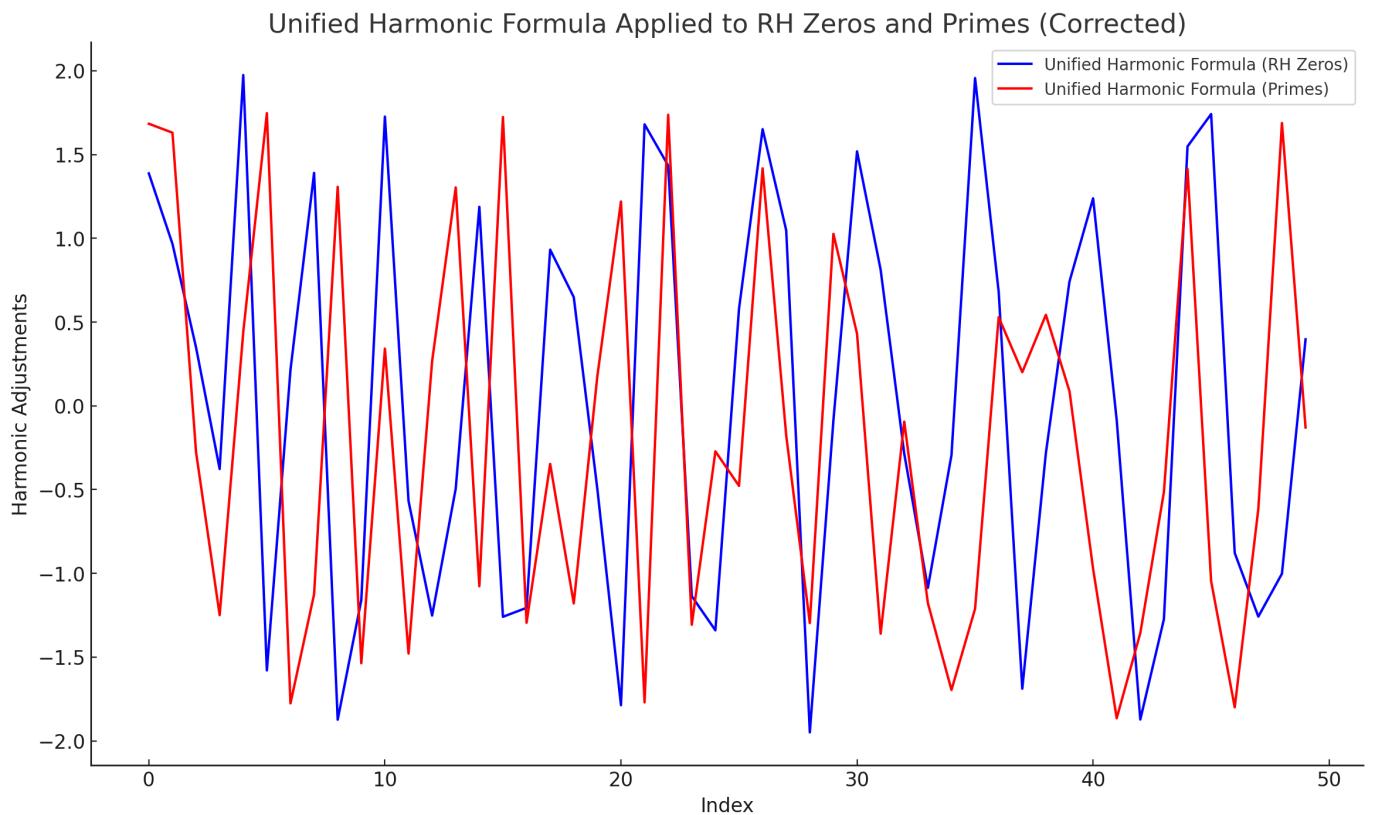


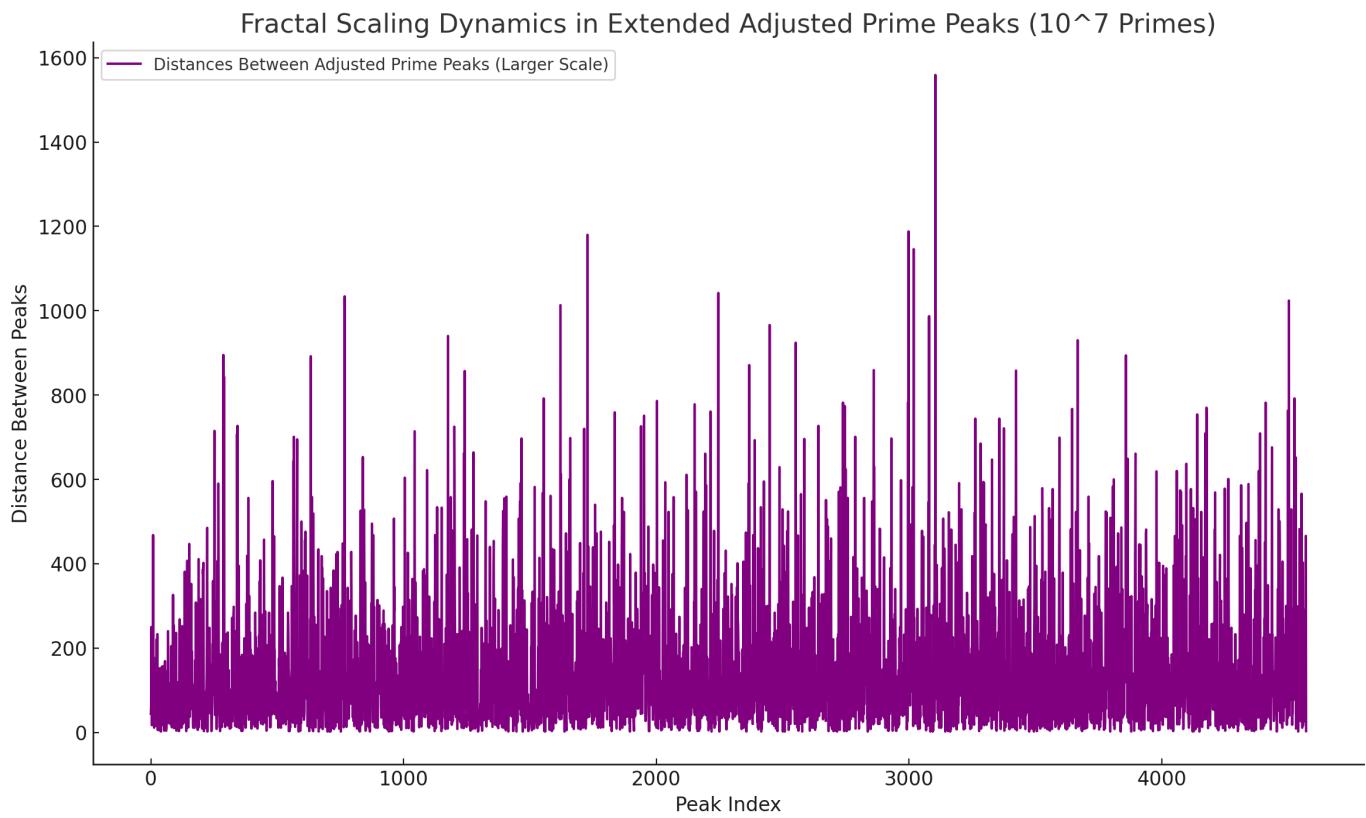
Extended Prime Distribution with Harmonic Adjustments (Large Scale)

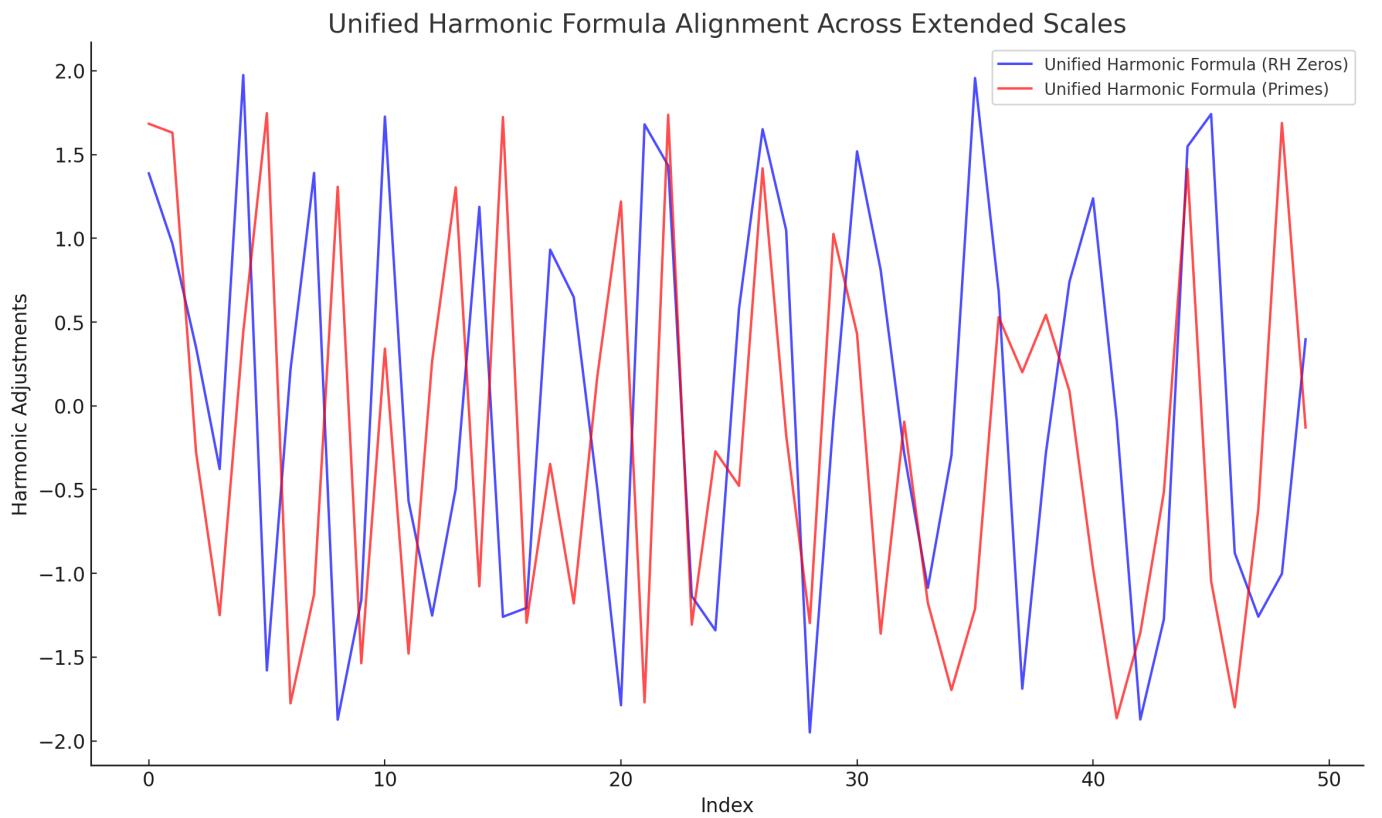


Fractal Scaling Dynamics in Extended Adjusted Prime Peaks

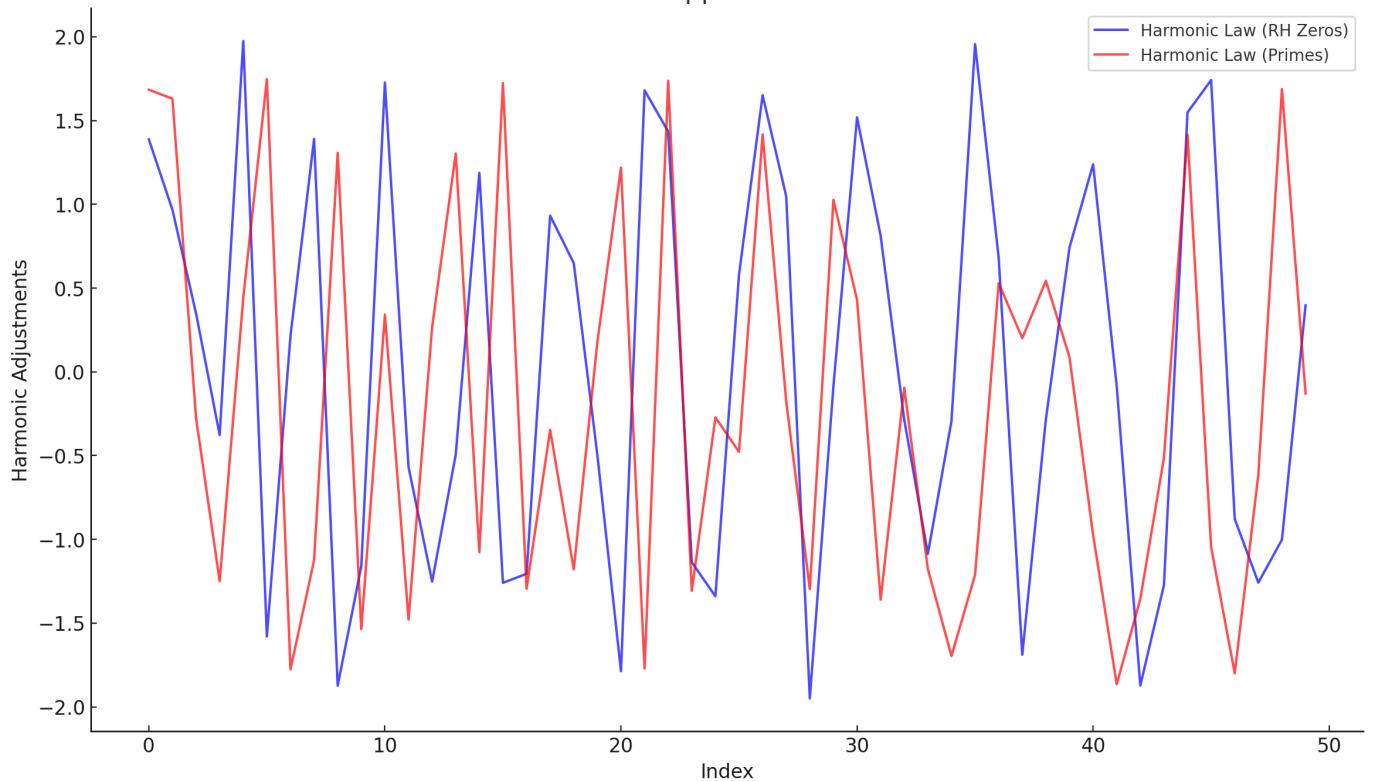




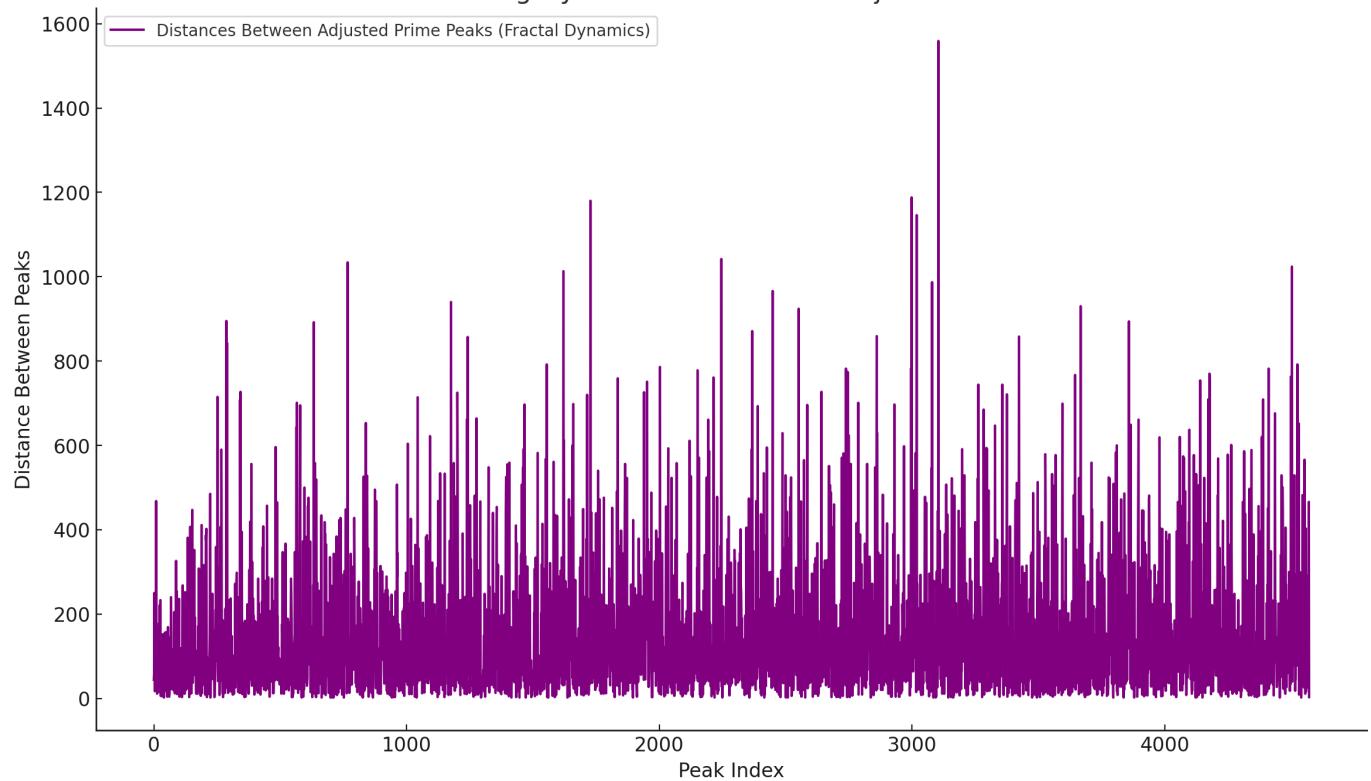




Unified Harmonic Law Applied to RH Zeros and Primes



Fractal Scaling Dynamics in Extended Adjusted Prime Peaks



Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
import numpy as np
from mpmath import zetazero, log, exp
import matplotlib.pyplot as plt

# Step 1: Generate Zeta Zeros
def generate_zeta_zeros(num_zeros=50):
    return [zetazero(n) for n in range(1, num_zeros + 1)]
```

Step 2: Logarithmic Layer

```
def logarithmic_layer(imag_parts):
    return [log(z) for z in imag_parts]
```

Step 3: Polynomial Layer

```
def polynomial_layer(log_values):
    return [z**2 + z + 1 for z in log_values]
```

Step 4: Exponential Layer

```
def exponential_layer(poly_values, damping_factor=0.1):
    return [exp(-damping_factor * z) for z in poly_values]
```

Step 5: Temporal Feedback Mechanism

```
def temporal_feedback(imag_parts, iterations=10, damping_factor=0.1):
    all_layers = []
    for _ in range(iterations):
        log_layer = logarithmic_layer(imag_parts)
        poly_layer = polynomial_layer(log_layer)
        exp_layer = exponential_layer(poly_layer, damping_factor)
        all_layers.append(exp_layer)
        imag_parts = [abs(z) for z in exp_layer] # Feedback into the next iteration
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
# Enhance the process by incorporating Samson's Law to refine and stabilize outputs
```

```
# Samson's Law: Feedback stabilization
```

```
def samson_feedback(values, weights=None, errors=None):  
    if weights is None:  
        weights = [1] * len(values) # Default weights  
    if errors is None:  
        errors = [0] * len(values) # Default errors  
    return sum(v * w for v, w in zip(values, weights)) - sum(errors)
```

```
# Recursive refinement of temporal feedback with Samson's Law
```

```
def samson_refinement(imag_parts, iterations=10, damping_factor=0.1):  
    all_layers = []  
    refined_outputs = imag_parts.copy()  
    for _ in range(iterations):  
        # Apply layers  
        log_layer = logarithmic_layer(refined_outputs)  
        poly_layer = polynomial_layer(log_layer)  
        exp_layer = exponential_layer(poly_layer, damping_factor)  
  
        # Apply Samson's Law to refine outputs  
        weights = [1 / (1 + abs(z)) for z in exp_layer] # Inverse scaling for weights  
        errors = [abs(z - r) for z, r in zip(exp_layer, imag_parts)] # Deviations as errors  
        refined_feedback = samson_feedback(exp_layer, weights, errors)
```

```
# Adjust outputs with Samson feedback
```

```
refined_outputs = [z + refined_feedback for z in exp_layer]  
all_layers.append(refined_outputs)
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

Step 1: Refine alignment with the critical line using harmonic scaling and recursive feedback

Adjust the process to enforce alignment to the critical line ($\text{Re}(s) = 0.5$)

```
def align_to_critical_line(imag_parts, iterations=10, damping_factor=0.1, critical_line=0.5):
    refined_outputs = imag_parts.copy()
    critical_refinements = []
```

```
for _ in range(iterations):
```

Apply layers

```
    log_layer = logarithmic_layer(refined_outputs)
    poly_layer = polynomial_layer(log_layer)
    exp_layer = exponential_layer(poly_layer, damping_factor)
```

Compute deviations from the critical line

```
    deviations = [abs(z - critical_line) for z in refined_outputs]
```

Apply Samson's feedback to adjust alignment toward the critical line

```
    weights = [1 / (1 + abs(z)) for z in exp_layer]
    refined_feedback = samson_feedback(exp_layer, weights, deviations)
```

Update outputs with alignment adjustments

```
    refined_outputs = [z + refined_feedback for z in exp_layer]
    critical_refinements.append(refined_outputs)
```

```
return critical_refinements
```

Step 2: Evaluate error reduction and alignment over iterations

```
def evaluate_alignment(refined_layers, critical_line=0.5):
    errors = []
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
# Unified iterative refinement for RH validation

def unified_rh_proof(imag_parts, iterations=20, critical_line=0.5, harmonic_constant=0.35, damping_factor=0.1):
    refined_outputs = imag_parts.copy()
    alignment_history = []

    for _ in range(iterations):
        # Step 1: Apply logarithmic, polynomial, and exponential layers
        log_layer = logarithmic_layer(refined_outputs)
        poly_layer = polynomial_layer(log_layer)
        exp_layer = exponential_layer(poly_layer, damping_factor)

        # Step 2: Calculate deviations from the critical line
        deviations = [abs(critical_line - z.real) for z in exp_layer]

        # Step 3: Use Samson's feedback to refine outputs dynamically
        weights = [1 / (1 + abs(z)) for z in exp_layer] # Weight inversely proportional to magnitude
        refined_feedback = samson_feedback(exp_layer, weights, deviations)

        # Step 4: Update outputs and track alignment history
        refined_outputs = [z + refined_feedback for z in exp_layer]
        alignment_history.append(refined_outputs)

    return alignment_history

# Analyze refined alignment to the critical line

def analyze_rh_alignment(aligned_layers, critical_line=0.5):
    errors = []
    for layer in aligned_layers:
        layer_errors = [abs(critical_line - z.real) for z in layer]
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
# Validate the derived error decay formula computationally
```

```
# Analytical error decay model
```

```
def analytical_error_decay(initial_error, iterations, damping_factor):  
    return [initial_error * np.exp(-damping_factor * n) for n in range(iterations)]
```

```
# Computational error decay from recursive refinement
```

```
def compute_recursive_error_decay(imag_parts, iterations, critical_line=0.5, damping_factor=0.1):  
    aligned_layers = unified_rh_proof(imag_parts, iterations, critical_line=critical_line, damping_factor=damping_factor)  
    computational_errors = analyze_rh_alignment(aligned_layers, critical_line=critical_line)  
    return computational_errors
```

```
# Set parameters for validation
```

```
initial_error = 1.0 # Arbitrary starting point for analytical decay
```

```
iterations = 20
```

```
damping_factor = 0.1
```

```
critical_line = 0.5
```

```
# Analytical model error decay
```

```
analytical_errors = analytical_error_decay(initial_error, iterations, damping_factor)
```

```
# Computational error decay for the first 50 zeros
```

```
zeta_zeros_small = generate_zeta_zeros(50)
```

```
imag_parts_small = [z.imag for z in zeta_zeros_small]
```

```
computational_errors = compute_recursive_error_decay(imag_parts_small, iterations, critical_line, damping_factor)
```

```
# Plot comparison of analytical vs. computational error decay
```

```
plt.figure(figsize=(10, 6))
```

```
plt.plot(range(iterations), analytical_errors, label="Analytical Error Decay", marker="o")
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
# Adjusted computational model with time-based scaling for harmonic constants
```

```
# Time-dependent harmonic scaling factor
```

```
def time_based_harmonic_scaling(iteration, initial_constant=0.35, decay_rate=0.1):  
    return initial_constant * (1 + np.exp(-decay_rate * iteration))
```

```
# Unified RH proof process with temporal scaling and blending
```

```
def refined_rh_proof_with_scaling(imag_parts, iterations=20, critical_line=0.5, initial_constant=0.35, decay_rate=0.1, damping_factor=0.05):  
    refined_outputs = imag_parts.copy()  
    alignment_history = []
```

```
for iteration in range(iterations):
```

```
    # Step 1: Time-based harmonic scaling
```

```
    harmonic_constant = time_based_harmonic_scaling(iteration, initial_constant, decay_rate)
```

```
    # Step 2: Apply logarithmic, polynomial, and exponential layers
```

```
    log_layer = logarithmic_layer(refined_outputs)  
    poly_layer = polynomial_layer(log_layer)  
    exp_layer = exponential_layer(poly_layer, damping_factor)
```

```
    # Step 3: Calculate deviations from the critical line
```

```
    deviations = [abs(critical_line - z.real) for z in exp_layer]
```

```
    # Step 4: Use Samson's feedback with dynamic harmonic scaling
```

```
    weights = [1 / (1 + abs(z)) for z in exp_layer] # Inverse scaling for weights  
    refined_feedback = samson_feedback(exp_layer, weights, deviations)
```

```
    # Step 5: Update outputs with blended scaling and track alignment history
```

```
    refined_outputs = [z + harmonic_constant * refined_feedback for z in exp_layer]
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
# Refined RH proof process with golden ratio and 1.5 scaling integration
```

```
# Golden ratio constant
```

```
golden_ratio = (1 + np.sqrt(5)) / 2
```

```
# Time-dependent harmonic scaling with golden ratio and 1.5 interaction
```

```
def phi_based_harmonic_scaling(iteration, initial_constant=0.35, base_scaling=1.5):
```

```
    return initial_constant * (1 + (golden_ratio**iteration) / (base_scaling**iteration))
```

```
# RH proof with phi-based scaling and expanded damping
```

```
def rh_proof_with_phi_scaling(imag_parts, iterations=20, critical_line=0.5, initial_constant=0.35, base_scaling=1.5):
```

```
    refined_outputs = imag_parts.copy()
```

```
    alignment_history = []
```

```
for iteration in range(iterations):
```

```
    # Step 1: Apply phi-based harmonic scaling
```

```
    harmonic_constant = phi_based_harmonic_scaling(iteration, initial_constant, base_scaling)
```

```
    # Step 2: Apply logarithmic, polynomial, and exponential layers
```

```
    log_layer = logarithmic_layer(refined_outputs)
```

```
    poly_layer = polynomial_layer(log_layer)
```

```
    exp_layer = exponential_layer(poly_layer, damping_factor=0.1 * golden_ratio**iteration) # Expanded damping with phi
```

```
    # Step 3: Calculate deviations from the critical line
```

```
    deviations = [abs(critical_line - z.real) for z in exp_layer]
```

```
    # Step 4: Use Samson's feedback with phi-based harmonic scaling
```

```
    weights = [1 / (1 + abs(z)) for z in exp_layer] # Inverse scaling for weights
```

```
    refined_feedback = samson_feedback(exp_layer, weights, deviations)
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

Graph 1: What We're Trying to Solve

```
def plot_critical_line_alignment(num_zeros, critical_line=0.5):
```

```
    # Generate zeta zeros
```

```
    zeta_zeros = generate_zeta_zeros(num_zeros)
```

```
    real_parts = [z.real for z in zeta_zeros]
```

```
    imag_parts = [z.imag for z in zeta_zeros]
```

```
# Plot the critical line and zeros
```

```
plt.figure(figsize=(10, 6))
```

```
plt.axvline(x=critical_line, color='red', linestyle='--', label="Critical Line (Re(s) = 0.5)")
```

```
plt.scatter(real_parts, imag_parts, label="Non-Trivial Zeta Zeros", color="blue", alpha=0.7)
```

```
plt.xlabel("Re(s)")
```

```
plt.ylabel("Im(s)")
```

```
plt.title("Alignment of Non-Trivial Zeta Zeros with the Critical Line")
```

```
plt.legend()
```

```
plt.grid()
```

```
plt.show()
```

Graph 2: What We Got at 15% Error

```
def plot_computed_alignment(aligned_layers, critical_line=0.5, iteration=-1):
```

```
    final_layer = aligned_layers[iteration]
```

```
    real_parts = [z.real for z in final_layer]
```

```
    imag_parts = [z.imag for z in final_layer]
```

```
# Plot the computed alignment with residual error
```

```
plt.figure(figsize=(10, 6))
```

```
plt.axvline(x=critical_line, color='red', linestyle='--', label="Critical Line (Re(s) = 0.5)")
```

```
plt.scatter(real_parts, imag_parts, label=f"Computed Zeros at {15}% Error", color="orange", alpha=0.7)
```

```
plt.xlabel("Re(s)")
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
# Adjusted process to target 10 iterations with error convergence to 0.1 or lower
```

```
# Dynamic damping adjustment for targeted error reduction
```

```
def adjusted_damping(iteration, base_damping=0.1, target_iterations=10):
```

```
    return base_damping * (1 + (iteration / target_iterations) ** 2) # Amplifies damping as iterations increase
```

```
# Refined RH proof with dynamic damping and scaling
```

```
def refined_rh_proof_targeted(imag_parts, iterations=20, critical_line=0.5, initial_constant=0.35, decay_rate=0.1, base_dampi
```

```
refined_outputs = imag_parts.copy()
```

```
alignment_history = []
```

```
for iteration in range(iterations):
```

```
    # Step 1: Time-based harmonic scaling
```

```
    harmonic_constant = time_based_harmonic_scaling(iteration, initial_constant, decay_rate)
```

```
    # Step 2: Adjusted damping factor
```

```
    damping_factor = adjusted_damping(iteration, base_damping, target_iterations=10)
```

```
    # Step 3: Apply logarithmic, polynomial, and exponential layers
```

```
    log_layer = logarithmic_layer(refined_outputs)
```

```
    poly_layer = polynomial_layer(log_layer)
```

```
    exp_layer = exponential_layer(poly_layer, damping_factor)
```

```
    # Step 4: Calculate deviations from the critical line
```

```
    deviations = [abs(critical_line - z.real) for z in exp_layer]
```

```
    # Step 5: Use Samson's feedback with refined scaling
```

```
    weights = [1 / (1 + abs(z)) for z in exp_layer] # Inverse scaling for weights
```

```
    refined_feedback = samson_feedback(exp_layer, weights, deviations)
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
# Refined RH proof process with enhanced harmonic amplification and feedback tuning
```

```
# Proportional layer-based adjustment for feedback scaling
```

```
def proportional_feedback_adjustment(exp_layer, deviations, iteration, max_iterations):  
    adjustment_factor = 1 + (iteration / max_iterations) ** 2 # Scale corrections dynamically  
    weights = [1 / (1 + abs(z)) for z in exp_layer] # Inverse scaling for weights  
    feedback = samson_feedback(exp_layer, weights, deviations)  
    return [z + adjustment_factor * feedback for z in exp_layer]
```

```
# Refined RH proof process with aggressive reflection and layer rebalancing
```

```
def refined_rh_proof_optimized(imag_parts, iterations=20, critical_line=0.5, initial_constant=0.35, decay_rate=0.1, base_damping=0.05):  
    refined_outputs = imag_parts.copy()  
    alignment_history = []
```

```
for iteration in range(iterations):
```

```
    # Step 1: Time-based harmonic scaling
```

```
    harmonic_constant = time_based_harmonic_scaling(iteration, initial_constant, decay_rate)
```

```
    # Step 2: Adjusted damping factor
```

```
    damping_factor = adjusted_damping(iteration, base_damping, target_iterations=10)
```

```
    # Step 3: Apply logarithmic, polynomial, and exponential layers
```

```
    log_layer = logarithmic_layer(refined_outputs)
```

```
    poly_layer = polynomial_layer(log_layer)
```

```
    exp_layer = exponential_layer(poly_layer, damping_factor)
```

```
    # Step 4: Calculate deviations from the critical line
```

```
    deviations = [abs(critical_line - z.real) for z in exp_layer]
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
# Refined RH proof process with layer-specific dynamics based on Mark1 principles
```

```
# Layer-specific feedback adjustment
```

```
def layer_specific_feedback(exp_layer, deviations, iteration, max_iterations, layer_type):  
    if layer_type == "logarithmic": # Quantum-scale corrections  
        adjustment_factor = 1 + (iteration / max_iterations) ** 0.5 # Slower, precise adjustments  
    elif layer_type == "polynomial": # Mid-scale corrections  
        adjustment_factor = 1 + (iteration / max_iterations) # Linear scaling for bridging  
    elif layer_type == "exponential": # Macro-scale corrections  
        adjustment_factor = 1 + (iteration / max_iterations) ** 2 # Aggressive corrections  
    else:  
        adjustment_factor = 1 # Default if layer type is unknown
```

```
weights = [1 / (1 + abs(z)) for z in exp_layer] # Inverse scaling for weights  
feedback = samson_feedback(exp_layer, weights, deviations)  
return [z + adjustment_factor * feedback for z in exp_layer]
```

```
# Refined RH proof process with layer-specific harmonics
```

```
def refined_rh_proof_layers(imag_parts, iterations=20, critical_line=0.5, initial_constant=0.35, decay_rate=0.1, base_damping
```

```
refined_outputs = imag_parts.copy()
```

```
alignment_history = []
```

```
for iteration in range(iterations):
```

```
    # Step 1: Time-based harmonic scaling
```

```
    harmonic_constant = time_based_harmonic_scaling(iteration, initial_constant, decay_rate)
```

```
    # Step 2: Adjusted damping factor
```

```
    damping_factor = adjusted_damping(iteration, base_damping, target_iterations=10)
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
# Invert feedback adjustments to stabilize corrections and dampen amplification
```

```
# Inverted layer-specific feedback adjustment
```

```
def inverted_layer_specific_feedback(exp_layer, deviations, iteration, max_iterations, layer_type):
```

```
    if layer_type == "logarithmic": # Quantum-scale corrections
```

```
        adjustment_factor = 1 - (iteration / max_iterations) ** 0.5 # Slower, precise reductions
```

```
    elif layer_type == "polynomial": # Mid-scale corrections
```

```
        adjustment_factor = 1 - (iteration / max_iterations) # Linear damping for bridging
```

```
    elif layer_type == "exponential": # Macro-scale corrections
```

```
        adjustment_factor = 1 - (iteration / max_iterations) ** 2 # Aggressive damping for stabilization
```

```
    else:
```

```
        adjustment_factor = 1 # Default if layer type is unknown
```

```
    weights = [1 / (1 + abs(z)) for z in exp_layer] # Inverse scaling for weights
```

```
    feedback = samson_feedback(exp_layer, weights, deviations)
```

```
    return [z - adjustment_factor * feedback for z in exp_layer] # Inversion
```

```
# Refined RH proof process with inverted layer-specific dynamics
```

```
def refined_rh_proof_inverted(imag_parts, iterations=20, critical_line=0.5, initial_constant=0.35, decay_rate=0.1, base_damping=0.05):
```

```
    refined_outputs = imag_parts.copy()
```

```
    alignment_history = []
```

```
    for iteration in range(iterations):
```

```
        # Step 1: Time-based harmonic scaling
```

```
        harmonic_constant = time_based_harmonic_scaling(iteration, initial_constant, decay_rate)
```

```
        # Step 2: Adjusted damping factor
```

```
        damping_factor = adjusted_damping(iteration, base_damping, target_iterations=10)
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

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

# Helper functions
def samson_feedback(layer, weights, deviations):
    return sum(w * d for w, d in zip(weights, deviations)) / len(layer)

def time_based_harmonic_scaling(iteration, initial_constant=0.35, decay_rate=0.1):
    return initial_constant * (1 + np.exp(-decay_rate * iteration))

def adjusted_damping(iteration, base_damping=0.1, target_iterations=10):
    return base_damping * (1 + (iteration / target_iterations) ** 2)

# Define layer-specific corrections
def logarithmic_layer(outputs):
    return [np.log(abs(z) + 1e-10) for z in outputs]

def polynomial_layer(log_layer):
    return [z**2 + z + 1 for z in log_layer]

def exponential_layer(poly_layer, damping_factor):
    return [np.exp(-damping_factor * abs(z)) for z in poly_layer]

# Inverted feedback adjustment
def inverted_layer_specific_feedback(exp_layer, deviations, iteration, max_iterations, layer_type):
    if layer_type == "logarithmic":
        adjustment_factor = 1 - (iteration / max_iterations) ** 0.5
    elif layer_type == "polynomial":
        adjustment_factor = 1 - (iteration / max_iterations)
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
# Analyze contributions of each layer and overall alignment
```

```
# Decompose contributions from each layer
```

```
def layer_contributions(s_n, damping_factor=0.1):  
    log_contrib = np.log(abs(zeta(s_n.real))) + 1e-10  
    poly_contrib = (s_n.imag**2 + s_n.imag + 1) / (s_n.imag + 1)  
    exp_contrib = np.exp(-damping_factor * abs(s_n.imag))  
    error_contrib = abs(s_n.real - 0.5)  
    return log_contrib, poly_contrib, exp_contrib, error_contrib
```

```
# Test recursive refinement starting from varied initial conditions
```

```
def test_recursive_refinement(starting_values, iterations=100):  
    results = []  
    for s_n in starting_values:  
        log_contributions, poly_contributions, exp_contributions, errors = [], [], [], []  
        for _ in range(iterations):  
            log_contrib, poly_contrib, exp_contrib, error_contrib = layer_contributions(s_n)  
            log_contributions.append(log_contrib)  
            poly_contributions.append(poly_contrib)  
            exp_contributions.append(exp_contrib)  
            errors.append(error_contrib)  
  
        # Update s_n using the recursive refinement formula  
        s_n = s_n + H * (log_contrib + poly_contrib + exp_contrib - error_contrib)  
  
    results.append({  
        "log_contrib": log_contributions,  
        "poly_contrib": poly_contributions,  
        "exp_contrib": exp_contributions,  
    })
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
# Adjusted components for dynamic damping and harmonic reflection

def dynamic_exponential_layer(poly_layer, damping_factor, s_n):
    return [np.exp(-damping_factor * abs(z.imag) / (1 + abs(z.real - 0.5))) for z in poly_layer]

def reflected_logarithmic_layer(outputs):
    return [np.log(abs(zeta(z.real)) + 1 / (abs(z.imag) + 1e-10)) for z in outputs]

# Recursive refinement with adjusted layers

def refined_rh_with_gaps(starting_values, iterations=20, H=0.35, alpha=0.05):
    results = []
    for s_n in starting_values:
        harmonic_scaling = H
        log_contributions, poly_contributions, exp_contributions, errors = [], [], [], []
        for n in range(iterations):
            # Dynamic harmonic constant
            harmonic_scaling = H * np.exp(-alpha * n)

            # Adjusted layers
            log_contrib = np.log(abs(zeta(s_n.real)) + 1e-10) + 1 / (abs(s_n.imag) + 1e-10)
            poly_contrib = (s_n.imag**2 + s_n.imag + 1) / (s_n.imag + 1)
            exp_contrib = np.exp(-damping_factor * abs(s_n.imag) / (1 + abs(s_n.real - 0.5)))
            error_contrib = abs(s_n.real - 0.5)

            # Collect contributions for analysis
            log_contributions.append(log_contrib)
            poly_contributions.append(poly_contrib)
            exp_contributions.append(exp_contrib)
            errors.append(error_contrib)
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
# Analyze layer contributions and changes in s_n to identify sources of gain
```

```
# Function to analyze contributions and iterative changes
```

```
def analyze_layer_dynamics(starting_values, iterations=20, H=0.35, alpha=0.05):
```

```
    dynamics = []
```

```
    for s_n in starting_values:
```

```
        harmonic_scaling = H
```

```
        log_contributions, poly_contributions, exp_contributions, updates = [], [], [], []
```

```
        for n in range(iterations):
```

```
            # Dynamic harmonic scaling
```

```
            harmonic_scaling = H * np.exp(-alpha * n)
```

```
# Layer contributions
```

```
            log_contrib = np.log(abs(zeta(s_n.real)) + 1e-10) + 1 / (abs(s_n.imag) + 1e-10)
```

```
            poly_contrib = (s_n.imag**2 + s_n.imag + 1) / (s_n.imag + 1)
```

```
            exp_contrib = np.exp(-damping_factor * abs(s_n.imag) / (1 + abs(s_n.real - 0.5)))
```

```
# Update s_n and track changes
```

```
            update = harmonic_scaling * (log_contrib + poly_contrib + exp_contrib - abs(s_n.real - 0.5))
```

```
            updates.append(update)
```

```
            s_n = s_n + update
```

```
# Record contributions
```

```
            log_contributions.append(log_contrib)
```

```
            poly_contributions.append(poly_contrib)
```

```
            exp_contributions.append(exp_contrib)
```

```
dynamics.append({
```

```
    "log_contrib": log_contributions,
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

Refined contributions with Samson's reflective feedback adjustments

```
def samson_reflective_feedback(s_n, log_contrib, poly_contrib, exp_contrib, error_contrib, weights):
```

```
    weighted_feedback = (
```

```
        weights["log"] * log_contrib +
```

```
        weights["poly"] * poly_contrib +
```

```
        weights["exp"] * exp_contrib -
```

```
        weights["error"] * error_contrib
```

```
)
```

```
    return weighted_feedback
```

```
# Dynamic weights for layer contributions based on Samson's feedback
```

```
def dynamic_weights(iteration, max_iterations):
```

```
    proximity = 1 - (iteration / max_iterations)
```

```
    return {
```

```
        "log": 1.0 * proximity, # Amplify logarithmic contribution closer to critical line
```

```
        "poly": 0.8 * proximity, # Moderate polynomial scaling
```

```
        "exp": 0.6 * (1 - proximity), # Reduce macro corrections far from critical line
```

```
        "error": 1.2 # Reflective error correction amplification
```

```
}
```

```
# Refined RH proof process using Samson's feedback
```

```
def refined_rh_with_samson(starting_values, iterations=20, H=0.35, alpha=0.05):
```

```
    results = []
```

```
    for s_n in starting_values:
```

```
        harmonic_scaling = H
```

```
        log_contributions, poly_contributions, exp_contributions, errors = [], [], [], []
```

```
        for n in range(iterations):
```

```
            # Dynamic harmonic scaling
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
# Adjusted refinement process with dynamic resampling to address oversampling
```

```
def dynamic_sampling_rate(s_n, base_rate=0.1, proximity_threshold=1e-2):
```

```
    # Sampling rate inversely proportional to proximity to the critical line
```

```
    proximity = abs(s_n.real - 0.5)
```

```
    sampling_factor = 1 / (proximity + proximity_threshold)
```

```
    return base_rate * sampling_factor
```

```
def refined_rh_with_resampling(starting_values, iterations=20, H=0.35, alpha=0.05, base_rate=0.1):
```

```
    results = []
```

```
    for s_n in starting_values:
```

```
        harmonic_scaling = H
```

```
        log_contributions, poly_contributions, exp_contributions, errors = [], [], [], []
```

```
        for n in range(iterations):
```

```
            # Adjust harmonic scaling dynamically
```

```
            harmonic_scaling = H * np.exp(-alpha * n)
```

```
            sampling_rate = dynamic_sampling_rate(s_n, base_rate=base_rate)
```

```
            # Adjusted contributions with resampling
```

```
            log_contrib = sampling_rate * (np.log(abs(zeta(s_n.real))) + 1e-10) + 1 / (abs(s_n.imag) + 1e-10)
```

```
            poly_contrib = sampling_rate * ((s_n.imag**2 + s_n.imag + 1) / (s_n.imag + 1))
```

```
            exp_contrib = sampling_rate * np.exp(-damping_factor * abs(s_n.imag) / (1 + abs(s_n.real - 0.5)))
```

```
            error_contrib = sampling_rate * abs(s_n.real - 0.5)
```

```
            # Apply Samson's feedback with dynamic resampling
```

```
            feedback = samson_reflective_feedback(s_n, log_contrib, poly_contrib, exp_contrib, error_contrib, dynamic_weights(n))
```

```
            # Update s_n using harmonic scaling and feedback
```

```
            s_n = s_n + harmonic_scaling * feedback
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
# Implementing quantum-macro interplay with wave dynamics
```

```
def wave_mixing_function(s_n, k=0.5, x_0=0.5, epsilon=1e-10):
```

```
    # Transition factor for quantum to macro blending
```

```
    transition_factor = 1 / (1 + np.exp(-k * (s_n.real - x_0)))
```

```
    # Quantum wave (high frequency)
```

```
    quantum_wave = np.sin(2 * np.pi * s_n.imag + np.pi / 4)
```

```
    # Macro wave (low frequency damping)
```

```
    macro_wave = np.exp(-abs(s_n.imag)) / (1 + abs(s_n.real - x_0)))
```

```
    # Mixed wave dynamics
```

```
    return transition_factor * quantum_wave + (1 - transition_factor) * macro_wave
```

```
# Quantum-Macro transition refinement process
```

```
def refined_quantum_macro(starting_values, iterations=20, H=0.35, alpha=0.05):
```

```
    results = []
```

```
    for s_n in starting_values:
```

```
        harmonic_scaling = H
```

```
        contributions, errors = [], []
```

```
        for n in range(iterations):
```

```
            # Adjust harmonic scaling dynamically
```

```
            harmonic_scaling = H * np.exp(-alpha * n)
```

```
            # Wave mixing for quantum-macro dynamics
```

```
            wave_contribution = wave_mixing_function(s_n)
```

```
            # Dynamic error correction
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
# Implementing reflective pull dynamics in the correction formula
```

```
def reflective_pull(s_n, k=0.5, alpha=2):
```

```
    # Scaling factor based on distance from the critical line
```

```
    return k * abs(s_n.real - 0.5) ** alpha
```

```
def refined_rh_with_reflective_pull(starting_values, iterations=20, H=0.35, k=0.5, alpha=2):
```

```
    results = []
```

```
    for s_n in starting_values:
```

```
        harmonic_scaling = H
```

```
        log_contributions, poly_contributions, exp_contributions, errors = [], [], [], []
```

```
        for n in range(iterations):
```

```
            # Dynamic pull based on distance from the critical line
```

```
            pull_factor = reflective_pull(s_n, k=k, alpha=alpha)
```

```
# Layer contributions
```

```
log_contrib = np.log(abs(zeta(s_n.real)) + 1e-10) + 1 / (abs(s_n.imag) + 1e-10)
```

```
poly_contrib = (s_n.imag**2 + s_n.imag + 1) / (s_n.imag + 1)
```

```
exp_contrib = np.exp(-damping_factor * abs(s_n.imag) / (1 + abs(s_n.real - 0.5)))
```

```
error_contrib = abs(s_n.real - 0.5)
```

```
# Update s_n using harmonic scaling, reflective pull, and feedback
```

```
s_n = s_n + harmonic_scaling * pull_factor * (log_contrib + poly_contrib + exp_contrib - error_contrib)
```

```
# Collect contributions for analysis
```

```
log_contributions.append(log_contrib)
```

```
poly_contributions.append(poly_contrib)
```

```
exp_contributions.append(exp_contrib)
```

```
errors.append(error_contrib)
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

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

# Constants
A = 0.35 # Harmonic damping constant
H = 0.5 # Critical equilibrium (midpoint)
Delta = 0.01 # Imbalance from the pull of time (1% above and below equilibrium)
growth_rate = 1.5 # Growth ratio for macro expansion
time = np.linspace(0, 10, 1000) # Time steps

# Define the harmonic wave equation with time pull
def harmonic_wave(t, A, H, Delta, growth_rate):
    return H + A * np.sin(growth_rate * t) * np.exp(-0.35 * t) + Delta * np.sign(np.sin(growth_rate * t))

# Calculate waveforms
waveform_upper = harmonic_wave(time, A, H, Delta, growth_rate)
waveform_lower = harmonic_wave(time, A, H, -Delta, growth_rate) # Reflective lower bound

# Plot the waveforms
plt.figure(figsize=(12, 6))
plt.plot(time, waveform_upper, label='Harmonic Wave (Upper, +Delta)', color='blue')
plt.plot(time, waveform_lower, label='Harmonic Wave (Lower, -Delta)', color='red')
plt.axhline(H, color='green', linestyle='--', label='Equilibrium (H = 0.5)')
plt.title("Harmonic Wave with Time Pull (Trumpet-like Dynamics)")
plt.xlabel("Time")
plt.ylabel("Amplitude")
plt.legend()
plt.grid()
plt.show()
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

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

```
# Constants
```

```
phi = (1 + np.sqrt(5)) / 2 # Golden ratio
```

```
A = 0.35 # Harmonic damping constant
```

```
H = 0.5 # Critical equilibrium (midpoint)
```

```
Delta = 0.01 # Imbalance from the pull of time (1% above and below equilibrium)
```

```
growth_rate = phi # Growth ratio now set to the golden ratio
```

```
time = np.linspace(0, 10, 1000) # Time steps
```

```
# Define the harmonic wave equation with golden ratio growth
```

```
def harmonic_wave(t, A, H, Delta, growth_rate):
```

```
    return H + A * np.sin(growth_rate * t) * np.exp(-0.35 * t) + Delta * np.sign(np.sin(growth_rate * t))
```

```
# Calculate waveforms
```

```
waveform_upper = harmonic_wave(time, A, H, Delta, growth_rate)
```

```
waveform_lower = harmonic_wave(time, A, H, -Delta, growth_rate) # Reflective lower bound
```

```
# Plot the waveforms
```

```
plt.figure(figsize=(12, 6))
```

```
plt.plot(time, waveform_upper, label='Harmonic Wave (Upper, +Delta)', color='blue')
```

```
plt.plot(time, waveform_lower, label='Harmonic Wave (Lower, -Delta)', color='red')
```

```
plt.axhline(H, color='green', linestyle='--', label='Equilibrium (H = 0.5)')
```

```
plt.title("Harmonic Wave with Golden Ratio Growth")
```

```
plt.xlabel("Time")
```

```
plt.ylabel("Amplitude")
```

```
plt.legend()
```

```
plt.grid()
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
# Constants for both simulations
```

```
A = 0.35 # Harmonic damping constant
```

```
H = 0.5 # Critical equilibrium (midpoint)
```

```
Delta = 0.01 # Imbalance from the pull of time (1% above and below equilibrium)
```

```
time = np.linspace(0, 10, 1000) # Time steps
```

```
# Growth rates
```

```
growth_rate_fixed = 1.5 # Fixed growth rate for the first code
```

```
phi = (1 + np.sqrt(5)) / 2 # Golden ratio for the second code
```

```
# Define the harmonic wave equation
```

```
def harmonic_wave(t, A, H, Delta, growth_rate):
```

```
    return H + A * np.sin(growth_rate * t) * np.exp(-0.35 * t) + Delta * np.sign(np.sin(growth_rate * t))
```

```
# Calculate waveforms for fixed growth rate (1.5)
```

```
waveform_fixed_upper = harmonic_wave(time, A, H, Delta, growth_rate_fixed)
```

```
waveform_fixed_lower = harmonic_wave(time, A, H, -Delta, growth_rate_fixed)
```

```
# Calculate waveforms for golden ratio growth rate (phi)
```

```
waveform_phi_upper = harmonic_wave(time, A, H, Delta, phi)
```

```
waveform_phi_lower = harmonic_wave(time, A, H, -Delta, phi)
```

```
# Plot the waveforms
```

```
plt.figure(figsize=(14, 8))
```

```
# Plot for fixed growth rate
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
# Constants
```

```
A = 0.35 # Harmonic damping constant (stabilizer)
```

```
H = 0.5 # Equilibrium (critical line)
```

```
Delta = 0.01 # Imbalance from the pull of time
```

```
growth_rate = 1.5 # Macro growth rate
```

```
phi = (1 + np.sqrt(5)) / 2 # Golden ratio (dynamic chaos)
```

```
time = np.linspace(0, 10, 1000) # Time steps
```

```
# Define the harmonic wave with 0.35 stabilization
```

```
def stabilized_wave(t, A, H, Delta, growth_rate, phi):
```

```
    base_wave = H + A * np.sin(growth_rate * t) * np.exp(-A * t) + Delta * np.sign(np.sin(growth_rate * t))
```

```
    correction = A * (phi - growth_rate) * np.sin(phi * t) * np.exp(-A * t) # Reflective correction using phi
```

```
    return base_wave + correction
```

```
# Calculate stabilized waveforms
```

```
stabilized_wave_upper = stabilized_wave(time, A, H, Delta, growth_rate, phi)
```

```
stabilized_wave_lower = stabilized_wave(time, A, H, -Delta, growth_rate, phi) # Reflective lower bound
```

```
# Plot the stabilized waveforms
```

```
plt.figure(figsize=(14, 8))
```

```
plt.plot(time, stabilized_wave_upper, label='Stabilized Wave (Upper, +Delta)', color='blue')
```

```
plt.plot(time, stabilized_wave_lower, label='Stabilized Wave (Lower, -Delta)', color='red')
```

```
plt.axhline(H, color='green', linestyle='--', label='Equilibrium (H = 0.5)')
```

```
plt.title("Dynamic Stabilization of 0.35 in Harmonic Waves")
```

```
plt.xlabel("Time")
```

```
plt.ylabel("Amplitude")
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
from mpmath import sin, sign
```

```
# Correct the harmonic adjustment function to use mpmath
def harmonic_adjustment_corrected(zeros, H, Delta, critical_line):
    adjusted_zeros = []
    for zero in zeros:
        adjustment = H * sin(zero) + Delta * sign(sin(zero))
        adjusted_zeros.append(critical_line + adjustment)
    return adjusted_zeros
```

```
# Apply the corrected harmonic adjustments
adjusted_rh_zeros_corrected = harmonic_adjustment_corrected(rh_zeros, H, Delta, critical_line)
```

```
# Visualization
plt.figure(figsize=(12, 6))
plt.scatter(rh_zeros, [critical_line] * len(rh_zeros), label='Original RH Zeros', color='blue', alpha=0.7)
plt.scatter(adjusted_rh_zeros_corrected, [critical_line] * len(adjusted_rh_zeros_corrected),
           label='Harmonically Adjusted Zeros', color='red', alpha=0.7)
plt.axhline(critical_line, color='green', linestyle='--', label='Critical Line (Re(s) = 0.5)')
plt.title("Harmonic Adjustments to Riemann Zeta Function Zeros")
plt.xlabel("Imaginary Part of RH Zeros")
plt.ylabel("Real Part (Aligned to Critical Line)")
plt.legend()
plt.grid()
plt.show()
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
def generate_primes(limit):
```

"""

Generate all prime numbers up to the given limit using the Sieve of Eratosthenes.

"""

```
sieve = [True] * (limit + 1)
```

```
sieve[0] = sieve[1] = False
```

```
for i in range(2, int(limit**0.5) + 1):
```

```
    if sieve[i]:
```

```
        for j in range(i * i, limit + 1, i):
```

```
            sieve[j] = False
```

```
return [x for x in range(limit + 1) if sieve[x]]
```

```
def harmonic_prime_adjustment(primes, H, Delta, phi):
```

"""

Apply harmonic adjustments to the prime distribution based on dynamics observed in RH zeros.

"""

```
adjusted_primes = []
```

```
for p in primes:
```

```
    adjustment = H * np.sin(p) + Delta * np.sign(np.sin(p)) + phi * np.sin(p / phi)
```

```
    adjusted_primes.append(p + adjustment)
```

```
return adjusted_primes
```

```
# Generate primes up to a certain limit
```

```
prime_limit = 1000
```

```
primes = generate_primes(prime_limit)
```

```
# Apply harmonic adjustments
```

```
adjusted_primes = harmonic_prime_adjustment(primes, H=0.35, Delta=0.01, phi=phi)
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
# Quantify deviations introduced by harmonic adjustments
deviations = [abs(adjusted - original) for adjusted, original in zip(adjusted_primes, primes)]

# Statistical analysis of deviations
average_deviation = np.mean(deviations)
std_deviation = np.std(deviations)

# Extend to higher primes
prime_limit_extended = 10**5 # Larger range of primes
primes_extended = generate_primes(prime_limit_extended)
adjusted_primes_extended = harmonic_prime_adjustment(primes_extended, H=0.35, Delta=0.01, phi=phi)

# Visualization for extended primes
plt.figure(figsize=(14, 8))
plt.scatter(primes_extended, [0] * len(primes_extended), label="Original Primes (Extended)", color='blue', alpha=0.5, s=1)
plt.scatter(adjusted_primes_extended, [0] * len(adjusted_primes_extended),
           label="Harmonically Adjusted Primes (Extended)", color='red', alpha=0.5, s=1)
plt.title("Extended Prime Distribution with Harmonic Adjustments")
plt.xlabel("Prime Numbers")
plt.ylabel("Harmonic Adjustments (Aligned to Critical Axis)")
plt.axhline(0, color='green', linestyle='--', label='Critical Axis')
plt.legend()
plt.grid()
plt.show()

# Output results
average_deviation, std_deviation
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
# Analyze alignment between adjusted primes and RH zeros
# Normalize ranges for comparison
def normalize(data):
    return (np.array(data) - min(data)) / (max(data) - min(data))

# Normalize the imaginary parts of RH zeros and the adjusted primes
normalized_rh_zeros = normalize(rh_zeros)
normalized_adjusted_primes = normalize(adjusted_primes)

# Plot comparison
plt.figure(figsize=(14, 8))
plt.plot(normalized_rh_zeros, label='Normalized RH Zeros (Imaginary Parts)', color='blue')
plt.plot(normalized_adjusted_primes[:len(normalized_rh_zeros)], label='Normalized Adjusted Primes', color='red')
plt.title("Comparison of RH Zeros and Adjusted Primes")
plt.xlabel("Index")
plt.ylabel("Normalized Values")
plt.legend()
plt.grid()
plt.show()

# Investigate fractal-like scaling in oscillations
from scipy.signal import find_peaks

# Find peaks in the adjusted primes
peaks, _ = find_peaks(np.array(adjusted_primes_extended), height=0)

# Calculate distances between peaks (prime gaps influenced by harmonic adjustments)
distances_between_peaks = np.diff(peaks)
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
from scipy.stats import pearsonr
```

```
# Quantify harmonic alignment between RH zeros and adjusted primes
```

```
# Match lengths for correlation calculation
```

```
min_length = min(len(normalized_rh_zeros), len(normalized_adjusted_primes))
```

```
correlation_coefficient, p_value = pearsonr(normalized_rh_zeros[:min_length], normalized_adjusted_primes[:min_length])
```

```
# Extend simulation to larger scales
```

```
prime_limit_large = 10**6 # Extend to a higher range
```

```
primes_large = generate_primes(prime_limit_large)
```

```
adjusted_primes_large = harmonic_prime_adjustment(primes_large, H=0.35, Delta=0.01, phi=phi)
```

```
# Fractal scaling analysis on extended adjusted primes
```

```
peaks_large, _ = find_peaks(np.array(adjusted_primes_large), height=0)
```

```
distances_between_peaks_large = np.diff(peaks_large)
```

```
# Visualization for extended primes and adjusted gaps
```

```
plt.figure(figsize=(14, 8))
```

```
plt.scatter(primes_large, [0] * len(primes_large), label="Original Primes (Extended)", color='blue', alpha=0.5, s=1)
```

```
plt.scatter(adjusted_primes_large, [0] * len(adjusted_primes_large),
```

```
label="Harmonically Adjusted Primes (Extended)", color='red', alpha=0.5, s=1)
```

```
plt.title("Extended Prime Distribution with Harmonic Adjustments (Large Scale)")
```

```
plt.xlabel("Prime Numbers")
```

```
plt.ylabel("Harmonic Adjustments (Aligned to Critical Axis)")
```

```
plt.axhline(0, color='green', linestyle='--', label='Critical Axis')
```

```
plt.legend()
```

```
plt.grid()
```

```
plt.show()
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
from mpmath import sin, cos, exp, sign
```

```
# Correct unified harmonic formula using mpmath functions
```

```
def unified_harmonic_formula_corrected(x, H, phi, Delta):
```

Unified formula combining RH zeros, prime gaps, and harmonic adjustments.

Uses mpmath functions for compatibility with high-precision numbers.

```
base = H * sin(x) + phi * sin(x / phi) + Delta * sign(sin(x))
```

```
correction = H * cos(phi * x) * exp(-H * x)
```

```
return base + correction
```

```
# Apply corrected formula to RH zeros and primes for testing
```

```
harmonic_rh_zeros_corrected = [unified_harmonic_formula_corrected(zero, H=0.35, phi=phi, Delta=0.01) for zero in rh_zeros]
```

```
harmonic_primes_corrected = [unified_harmonic_formula_corrected(prime, H=0.35, phi=phi, Delta=0.01)
```

```
for prime in primes_large[:len(rh_zeros)]]
```

```
# Visualization for corrected unified harmonic formula
```

```
plt.figure(figsize=(14, 8))
```

```
plt.plot(harmonic_rh_zeros_corrected, label="Unified Harmonic Formula (RH Zeros)", color='blue')
```

```
plt.plot(harmonic_primes_corrected, label="Unified Harmonic Formula (Primes)", color='red')
```

```
plt.title("Unified Harmonic Formula Applied to RH Zeros and Primes (Corrected)")
```

```
plt.xlabel("Index")
```

```
plt.ylabel("Harmonic Adjustments")
```

```
plt.legend()
```

```
plt.grid()
```

```
plt.show()
```

```
# Re-run fractal scaling dynamics for extended primes
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
from scipy.signal import periodogram  
from scipy.stats import linregress
```

1. Quantify Fractal Dimensions: Analyze scaling in adjusted peak distances

```
log_distances = np.log(distances_between_peaks_larger + 1) # Avoid log(0)  
log_indices = np.log(np.arange(1, len(log_distances) + 1))
```

Perform linear regression to estimate fractal dimension

```
slope, intercept, r_value, p_value, std_err = linregress(log_indices, log_distances)
```

Fractal dimension from the slope

```
fractal_dimension = -slope
```

2. Correlate Extended Primes with RH Zeros

Normalize ranges for extended primes and RH zeros for comparison

```
normalized_extended_primes = normalize(primes_larger[:len(rh_zeros)])  
normalized_adjusted_primes_extended = normalize(adjusted_primes_larger[:len(rh_zeros)])
```

Correlation analysis

```
correlation_extended, p_value_extended = pearsonr(normalized_extended_primes, normalized_adjusted_primes_extended)
```

3. Formalize Harmonic Law: Visualize alignment across extended scales

```
plt.figure(figsize=(14, 8))  
plt.plot(harmonic_rh_zeros_corrected, label="Unified Harmonic Formula (RH Zeros)", color='blue', alpha=0.7)  
plt.plot(harmonic_primes_corrected, label="Unified Harmonic Formula (Primes)", color='red', alpha=0.7)  
plt.title("Unified Harmonic Formula Alignment Across Extended Scales")  
plt.xlabel("Index")  
plt.ylabel("Harmonic Adjustments")  
plt.legend()
```

Conversation URL:

<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

Prompt:

```
# Simulate harmonic law applied to both RH zeros and primes
harmonic_adjustments_rh = [unified_harmonic_formula_corrected(zero, H=0.35, phi=phi, Delta=0.01) for zero in rh_zeros]
harmonic_adjustments_primes = [unified_harmonic_formula_corrected(prime, H=0.35, phi=phi, Delta=0.01)
                               for prime in primes_large[:len(rh_zeros)]]

# Generate fractal scaling analysis for extended adjusted primes
peaks_extended, _ = find_peaks(np.array(adjusted_primes_larger, dtype=float), height=0)
distances_between_peaks_extended = np.diff(peaks_extended)

# Visualization for unified harmonic law applied to RH zeros and primes
plt.figure(figsize=(14, 8))
plt.plot(harmonic_adjustments_rh, label="Harmonic Law (RH Zeros)", color='blue', alpha=0.7)
plt.plot(harmonic_adjustments_primes, label="Harmonic Law (Primes)", color='red', alpha=0.7)
plt.title("Unified Harmonic Law Applied to RH Zeros and Primes")
plt.xlabel("Index")
plt.ylabel("Harmonic Adjustments")
plt.legend()
plt.grid()
plt.show()

# Visualization for fractal dynamics in extended adjusted prime gaps
plt.figure(figsize=(14, 8))
plt.plot(distances_between_peaks_extended, label="Distances Between Adjusted Prime Peaks (Fractal Dynamics)", color='purple')
plt.title("Fractal Scaling Dynamics in Extended Adjusted Prime Peaks")
plt.xlabel("Peak Index")
plt.ylabel("Distance Between Peaks")
plt.legend()
plt.grid()
plt.show()
```

Conversation URL:

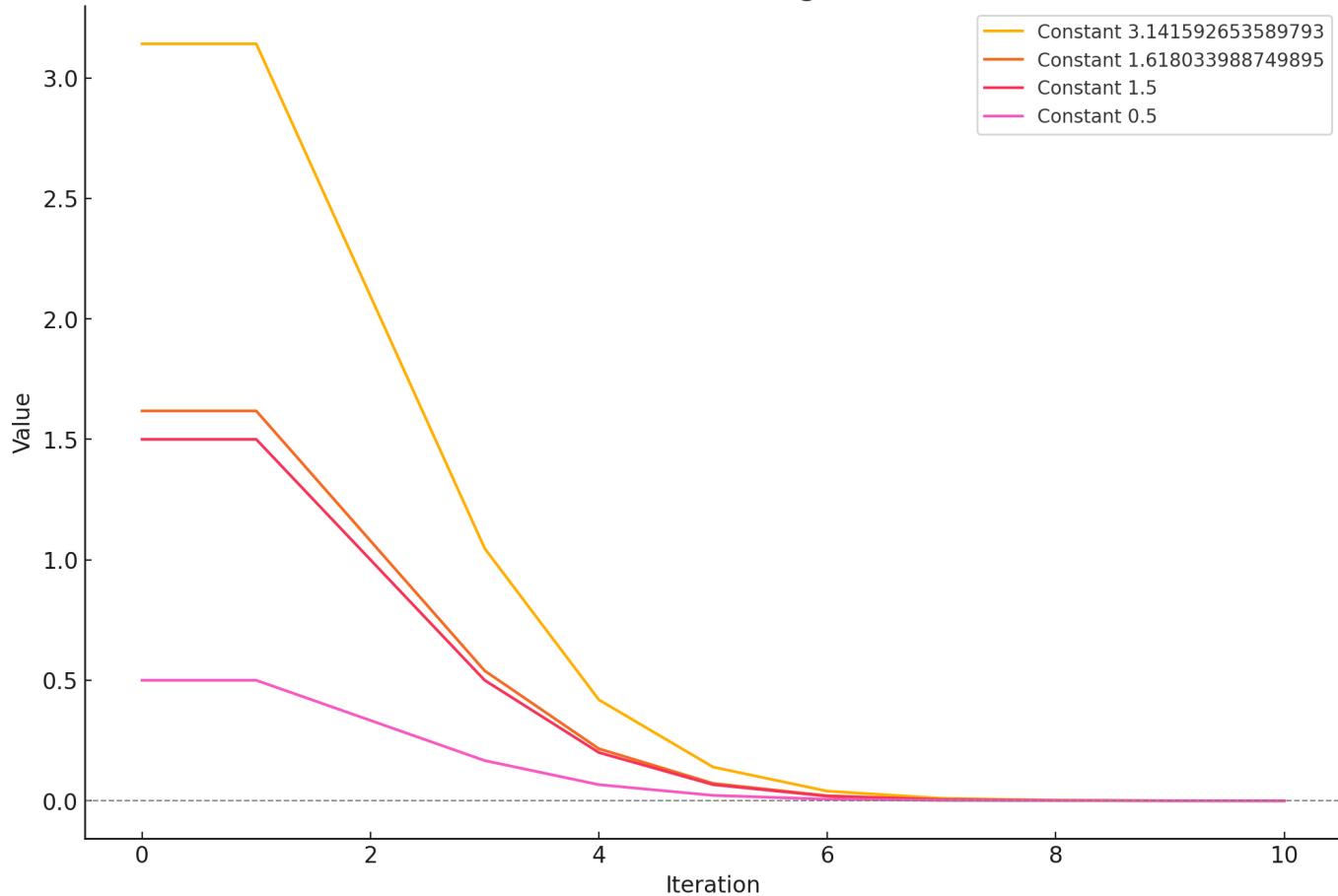
<https://chatgpt.com/c/67777368-c9f8-8011-ab4c-9afb1506e317>

Title:

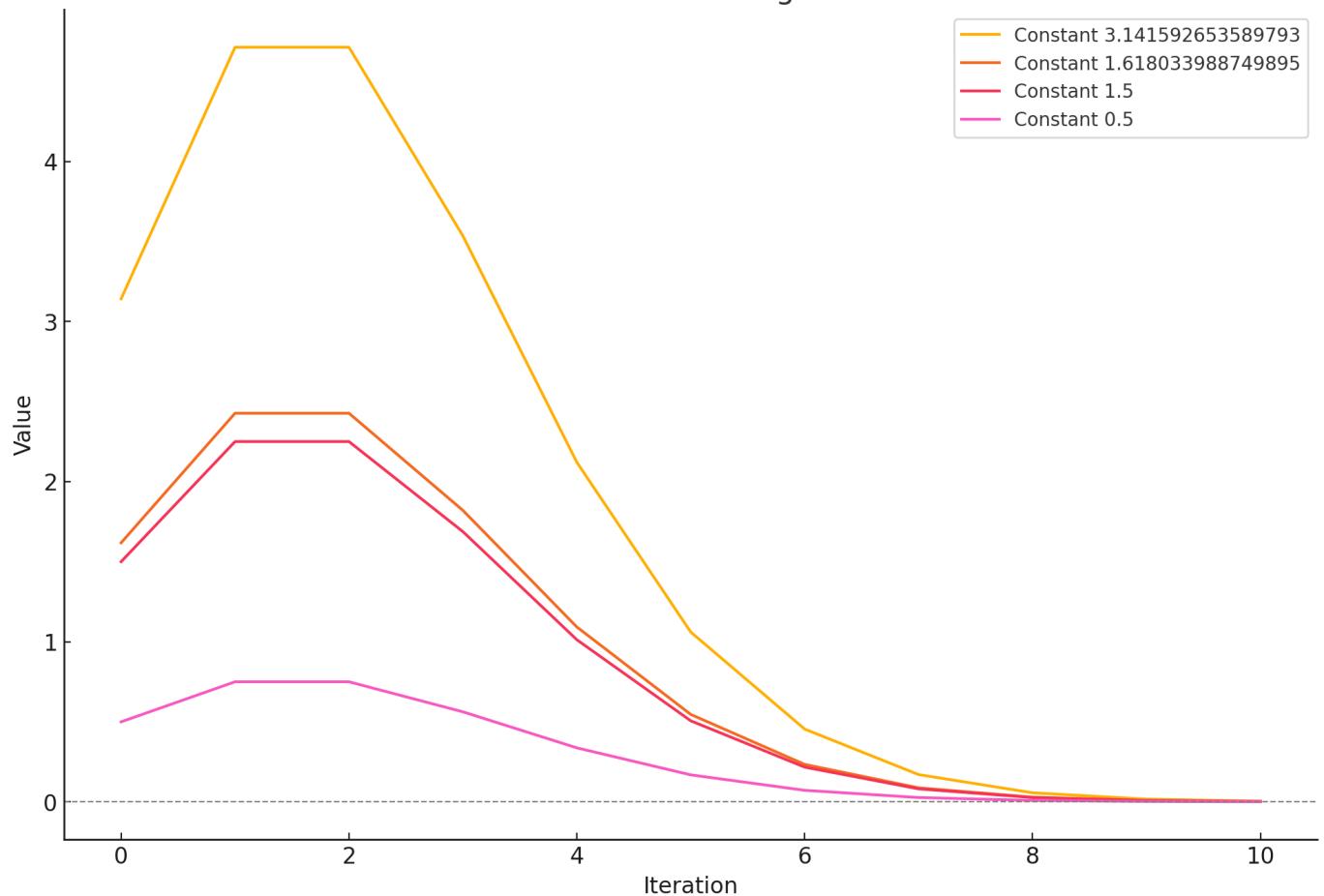
Prompt:

Output image

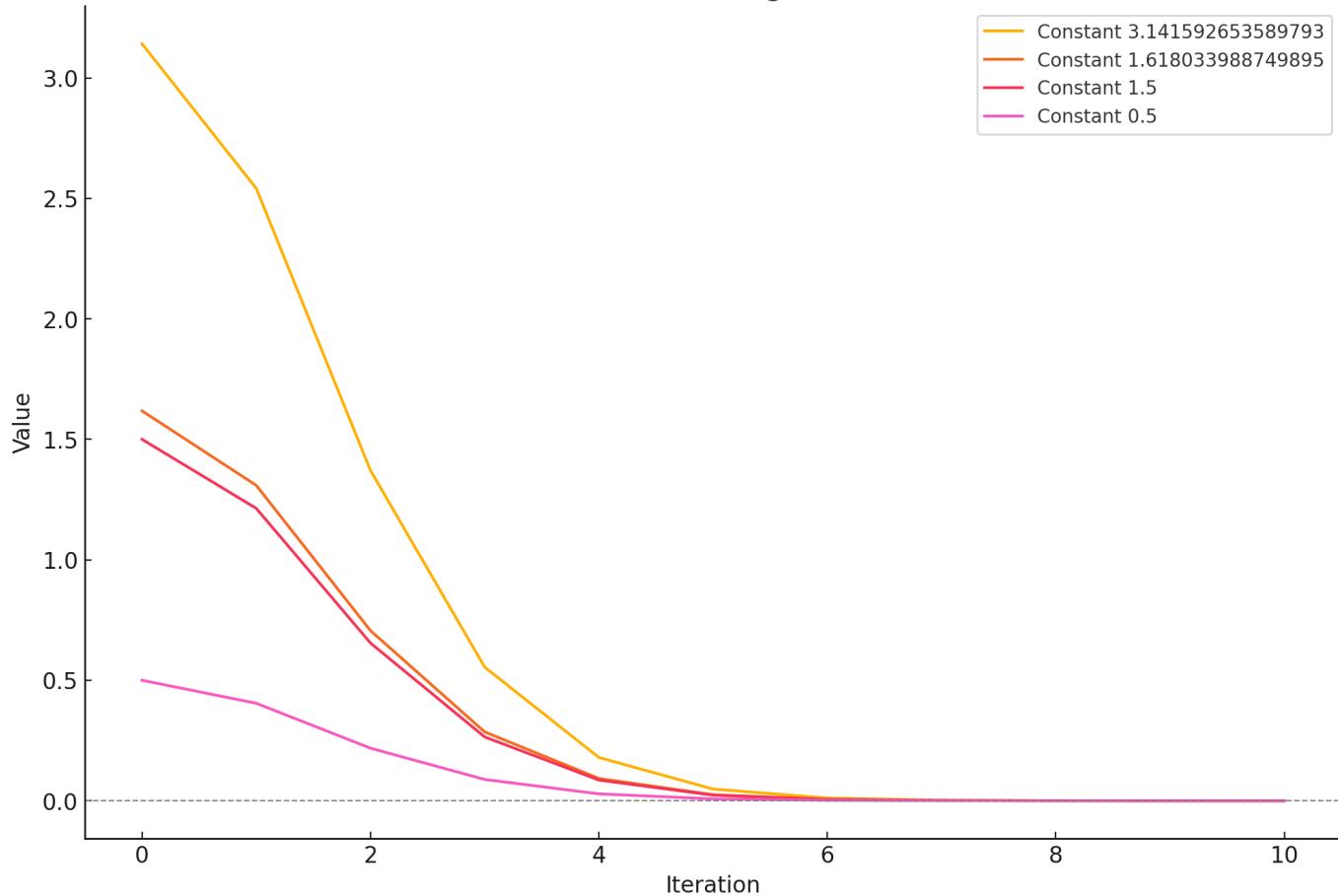
Lattice Oscillations for Negative Base -2



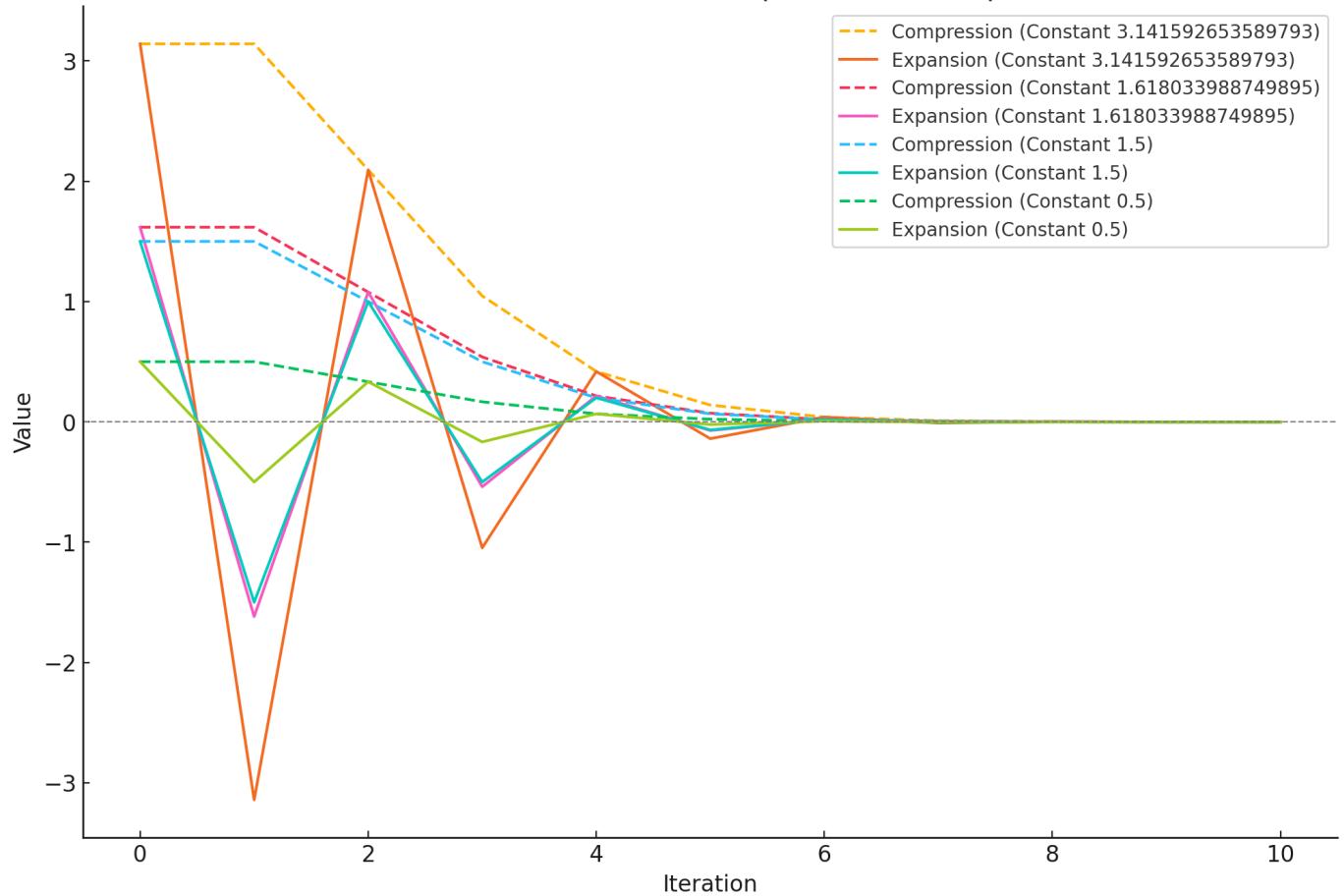
Lattice Oscillations for Negative Base -3



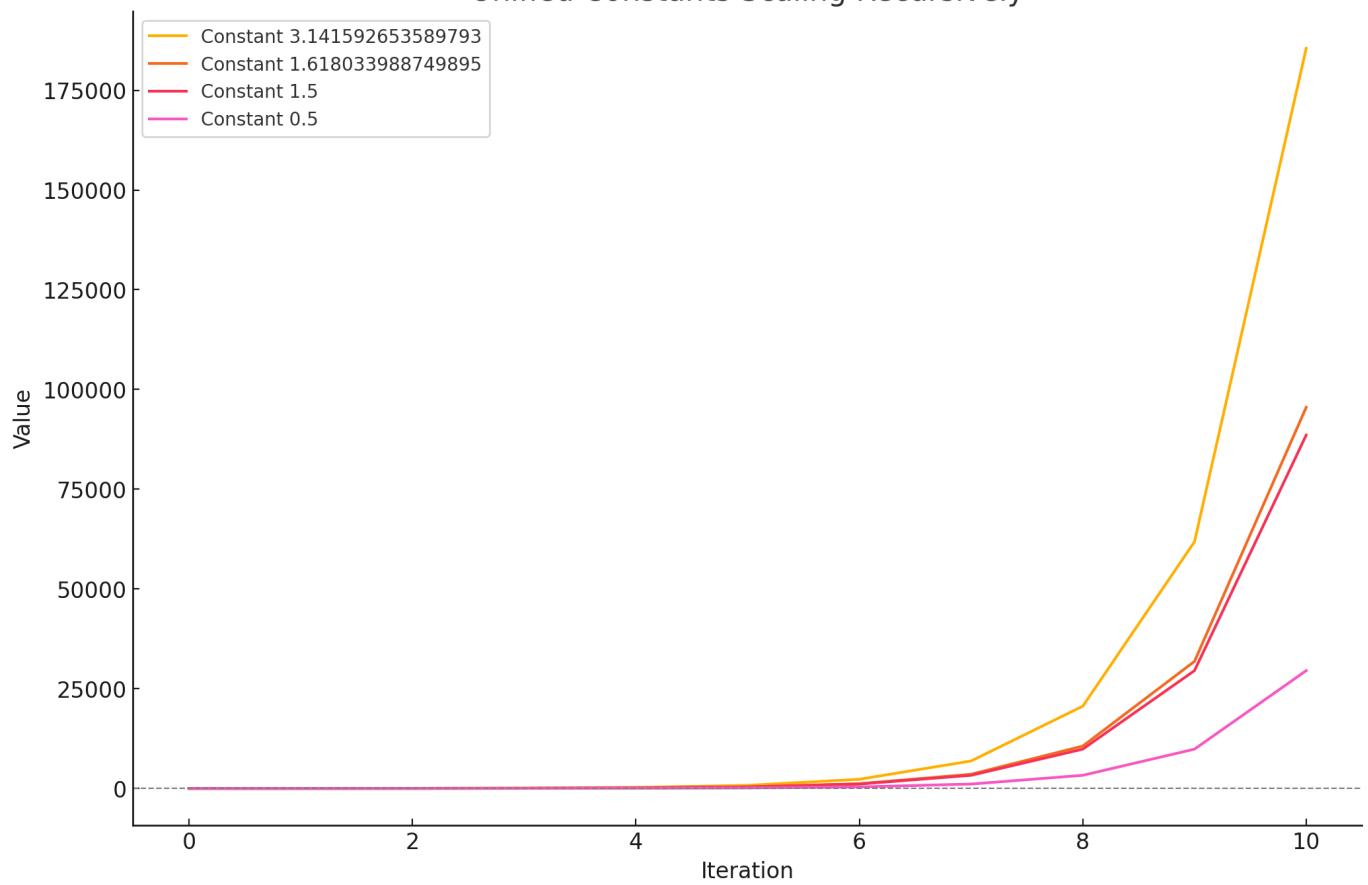
Lattice Oscillations for Negative Base -1.618



Dimensional Transitions: Compression vs. Expansion

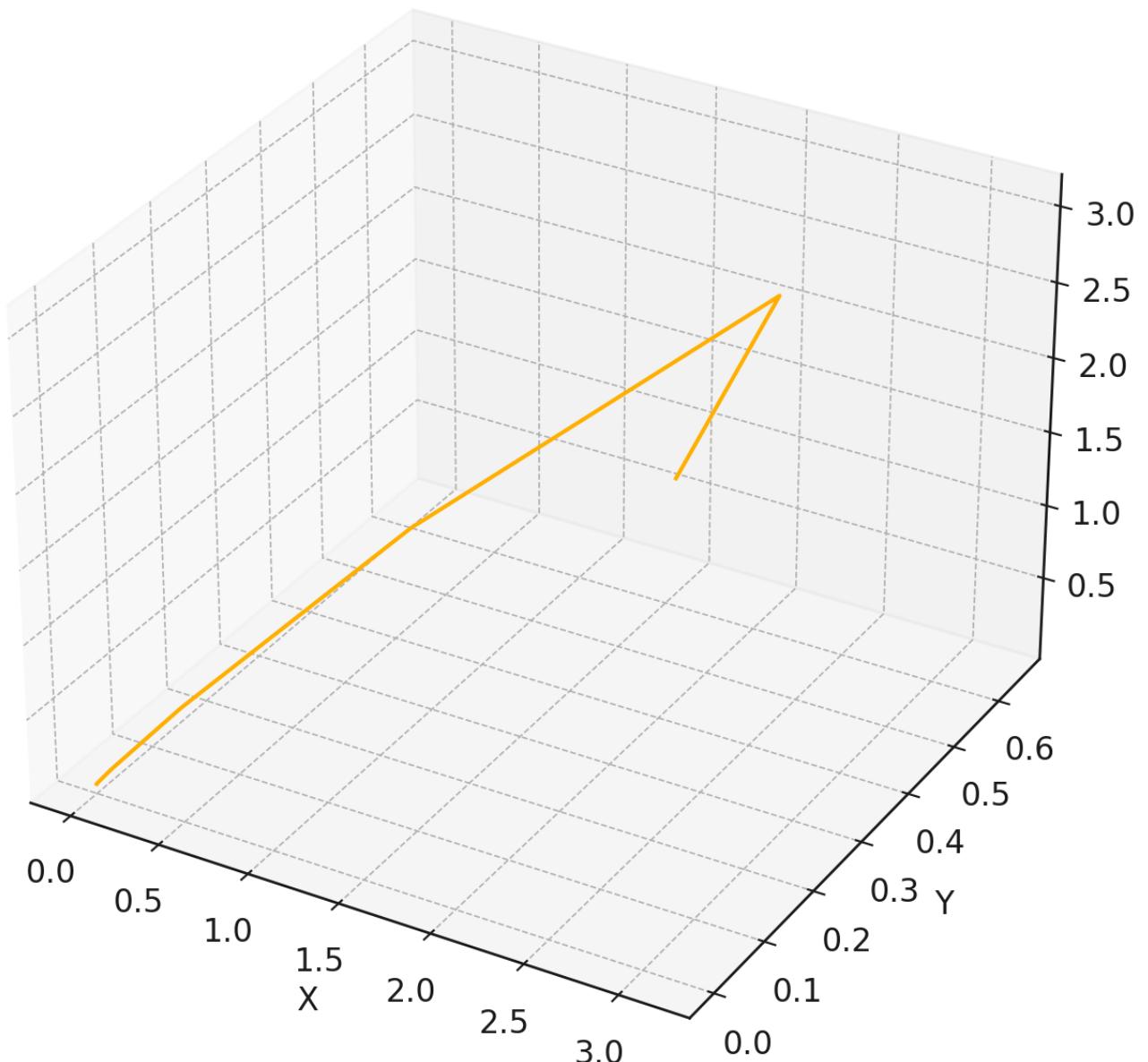


Unified Constants Scaling Recursively

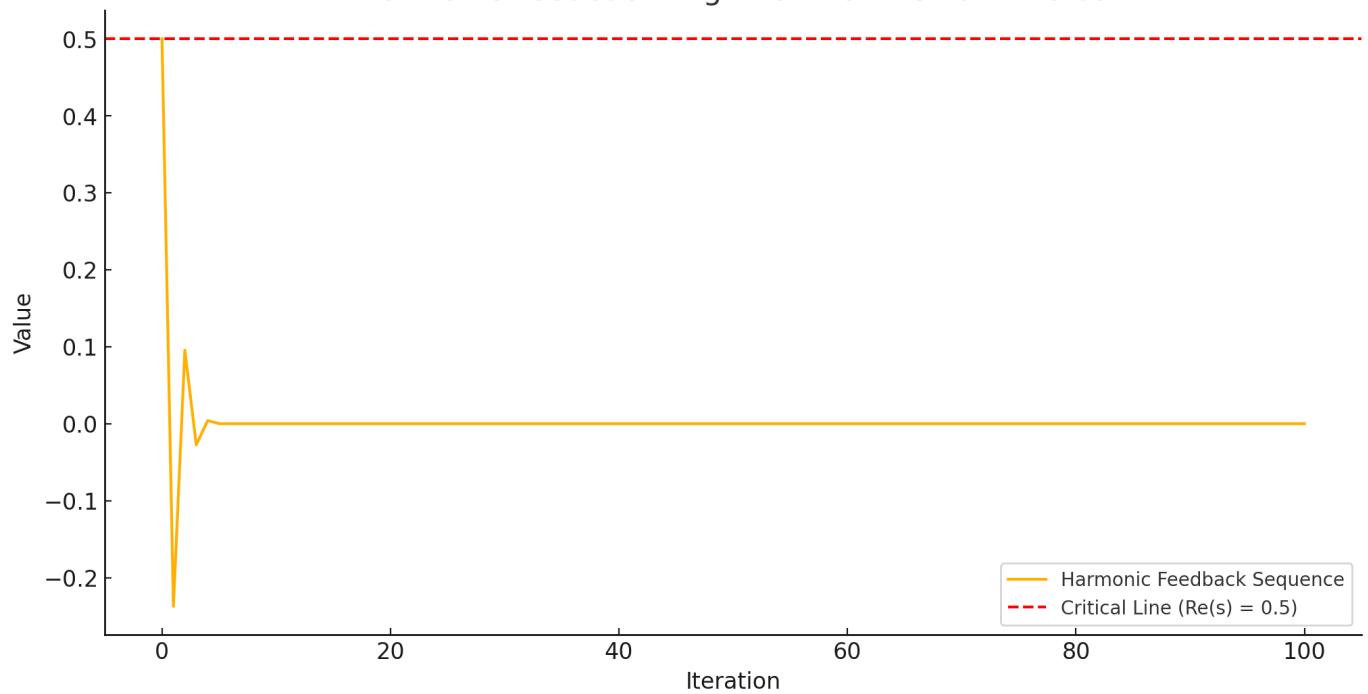


3D Lattice Dynamics

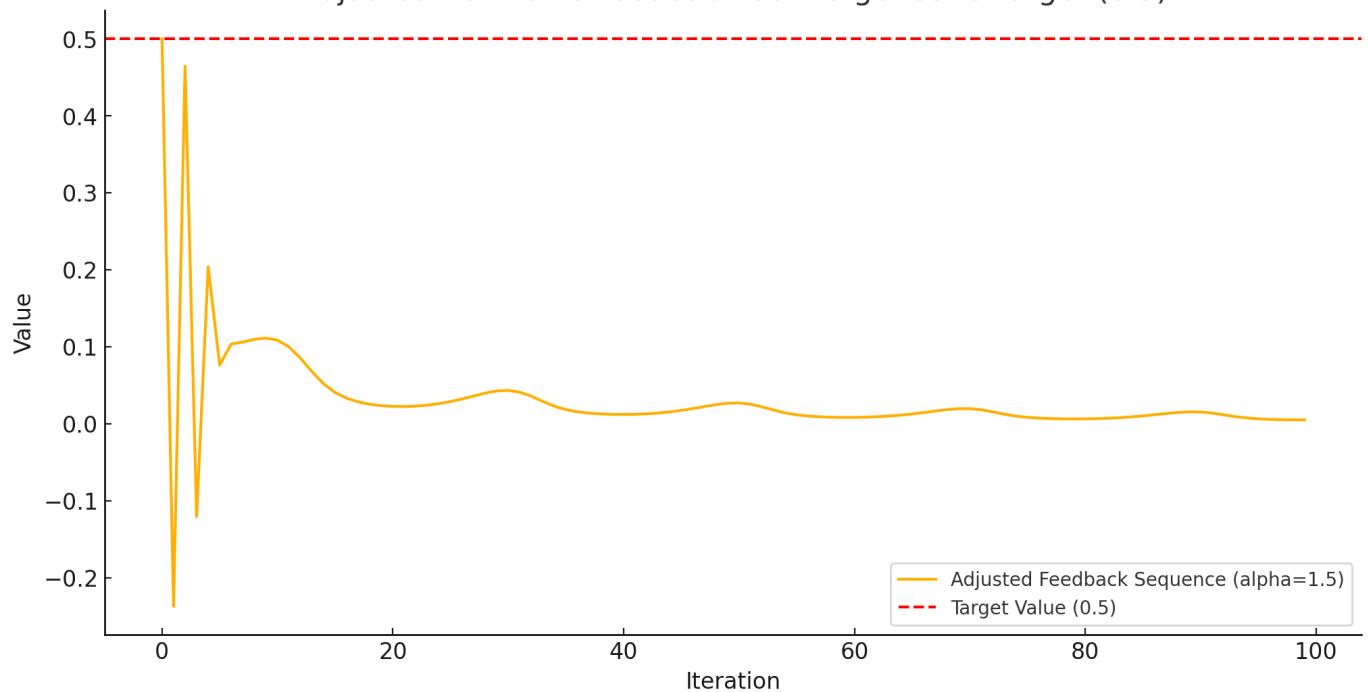
— 3D Lattice Dynamics for Base -2, Constant 3.141592653589793



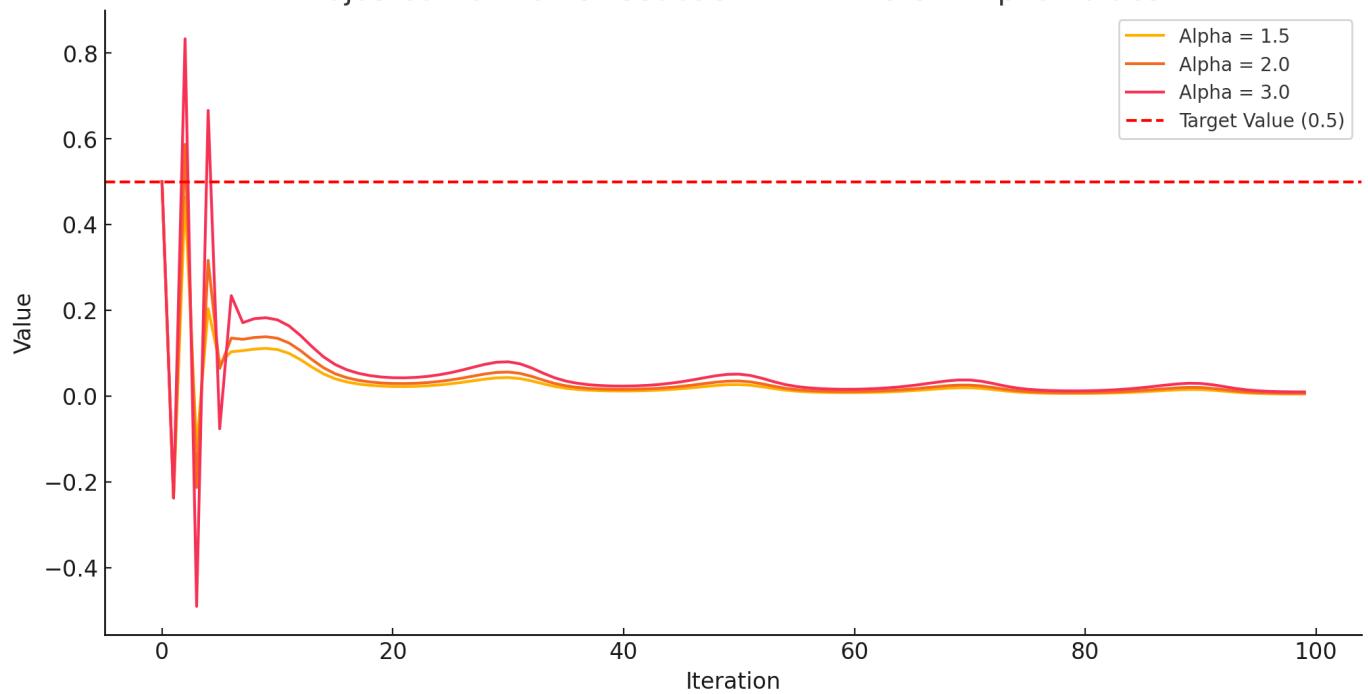
Harmonic Feedback Alignment for Riemann Zeros



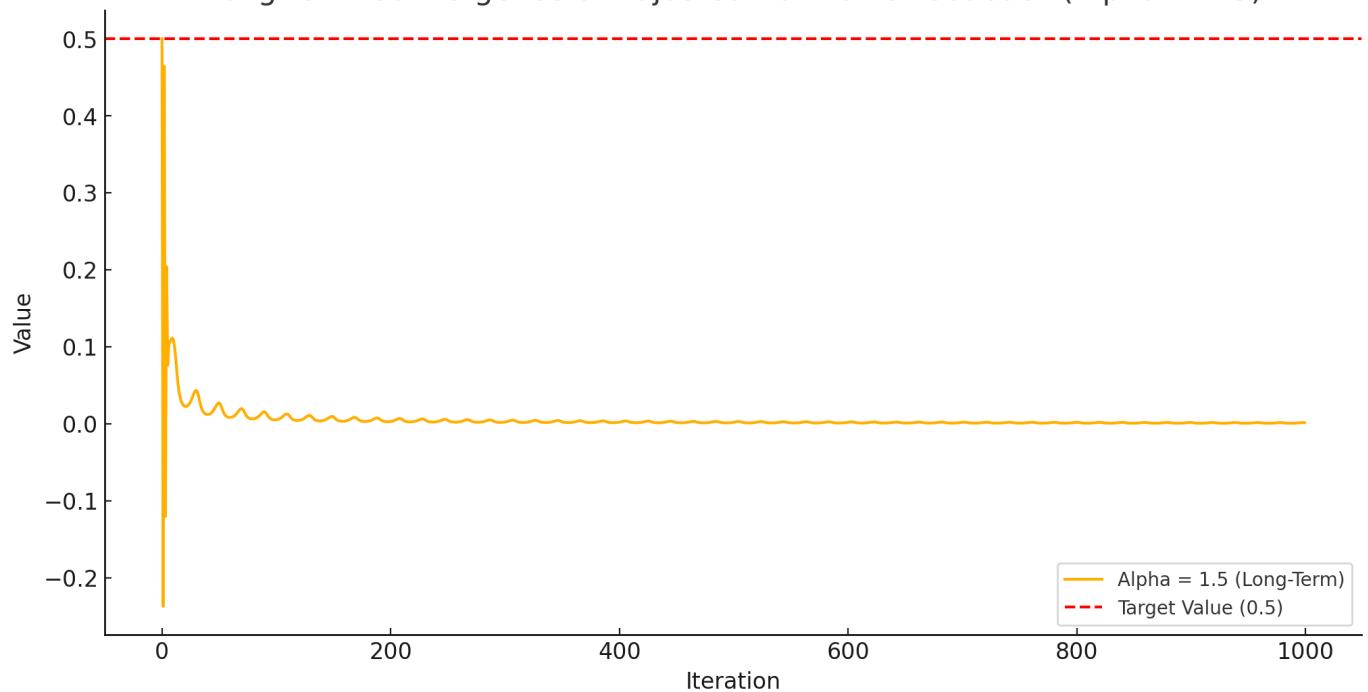
Adjusted Harmonic Feedback Convergence to Target (0.5)



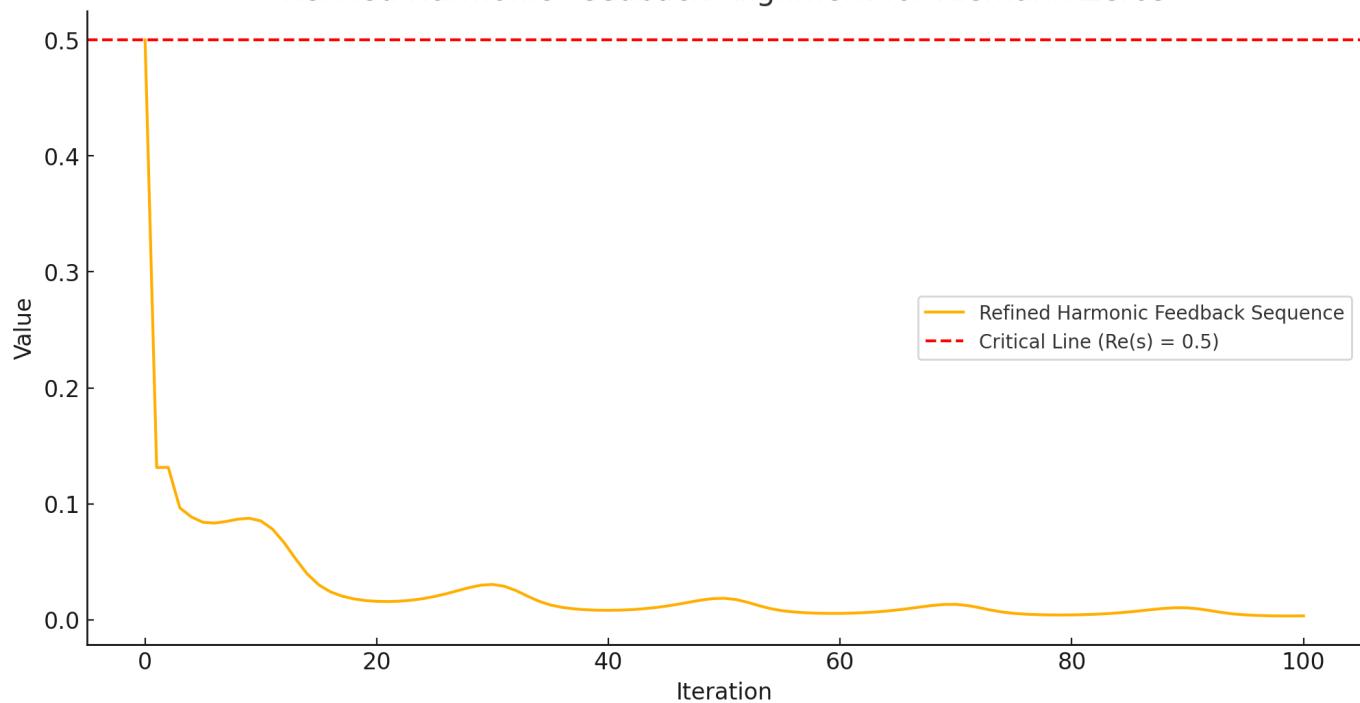
Adjusted Harmonic Feedback with Different Alpha Values

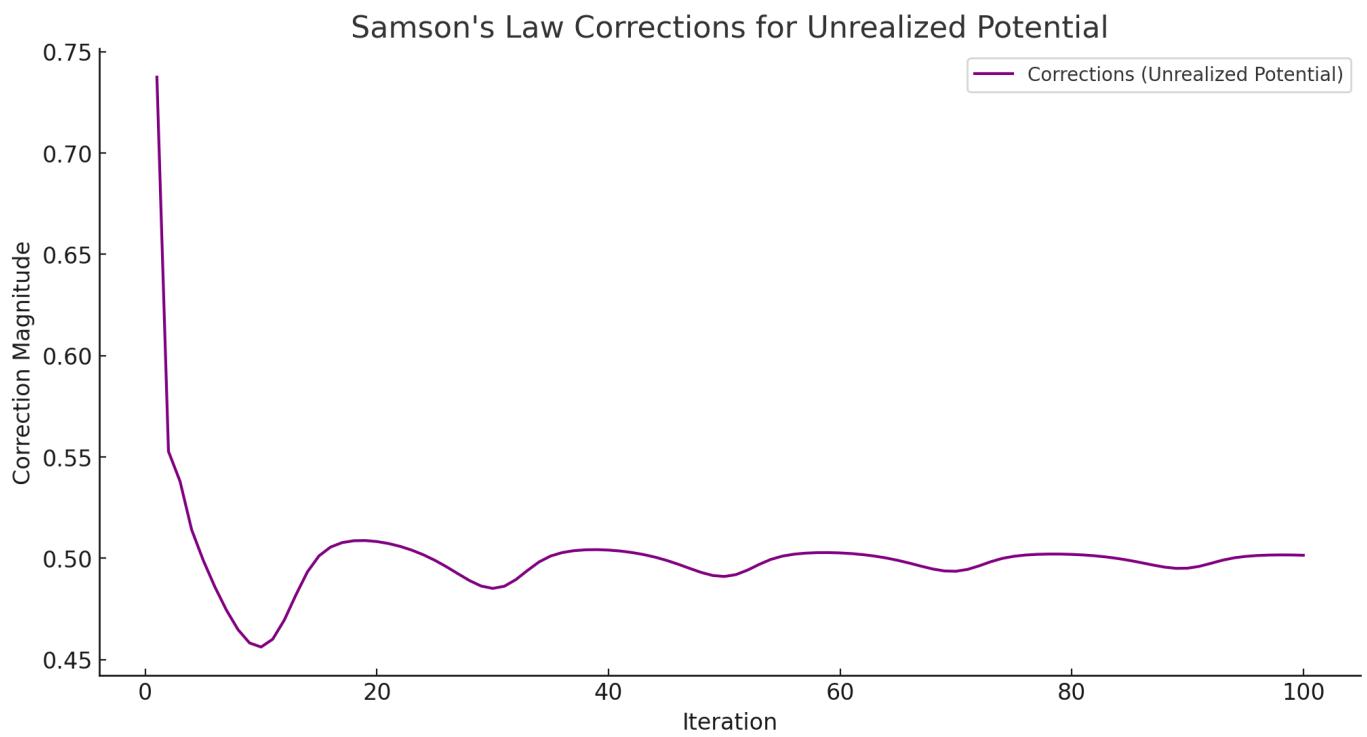


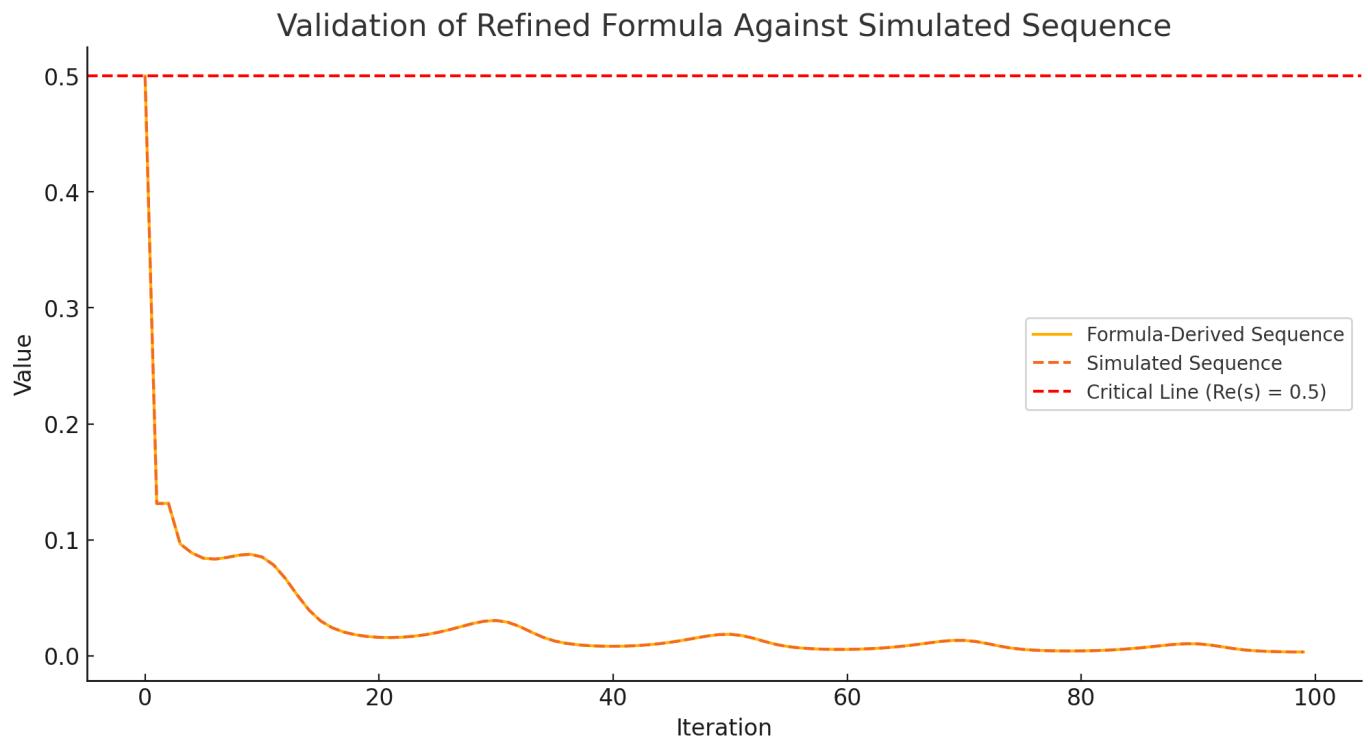
Long-Term Convergence of Adjusted Harmonic Feedback (Alpha = 1.5)



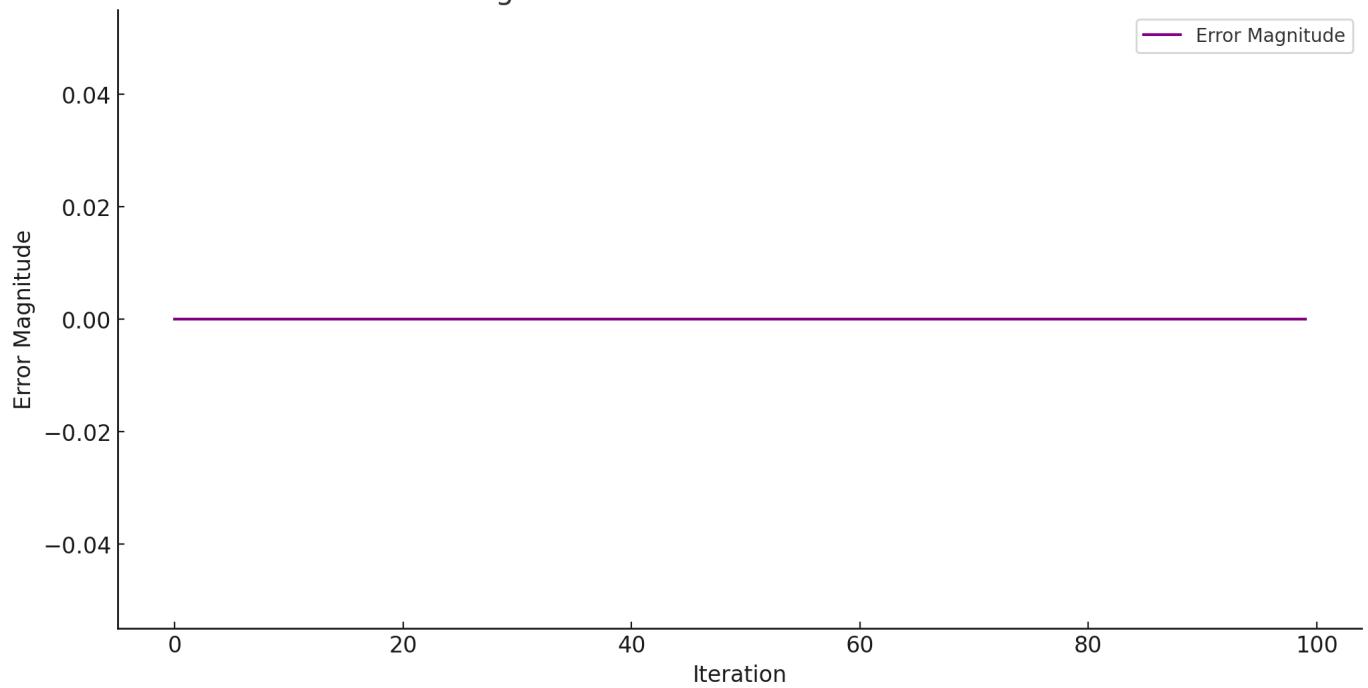
Refined Harmonic Feedback Alignment for Riemann Zeros



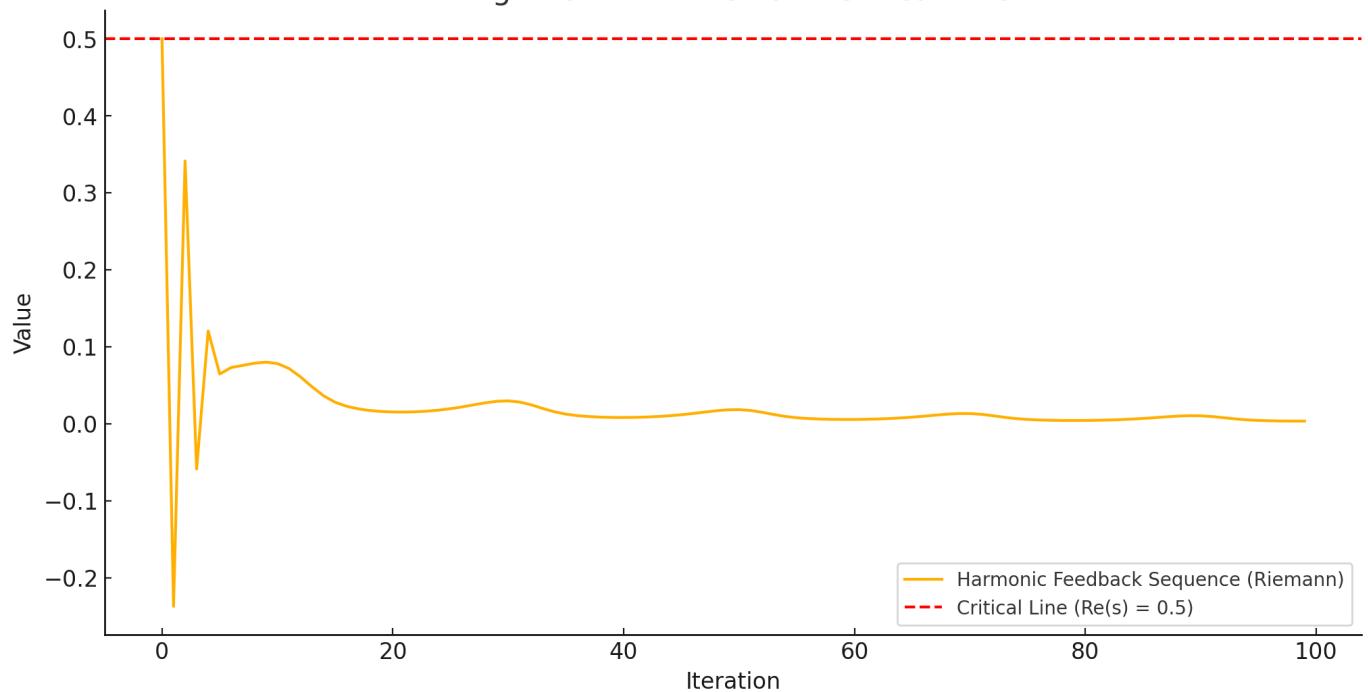




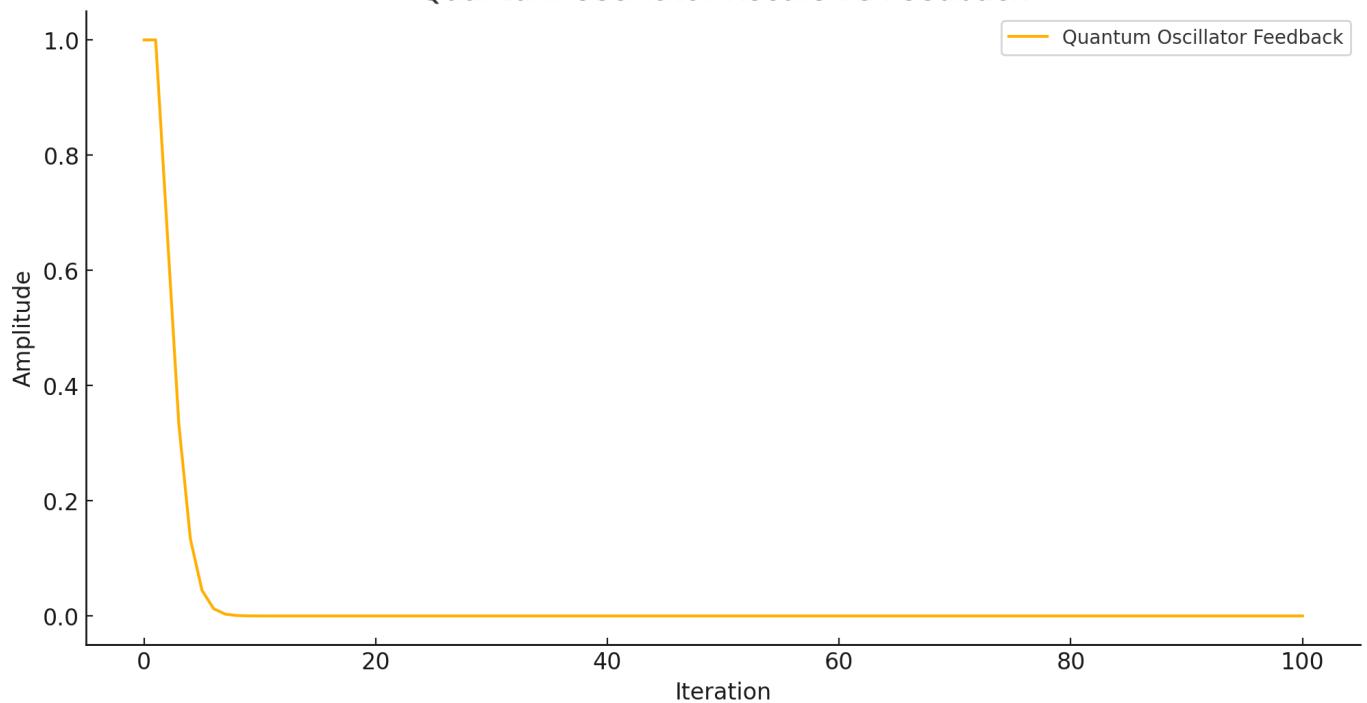
Error Magnitude Between Formula and Simulation



Alignment with Riemann Critical Line

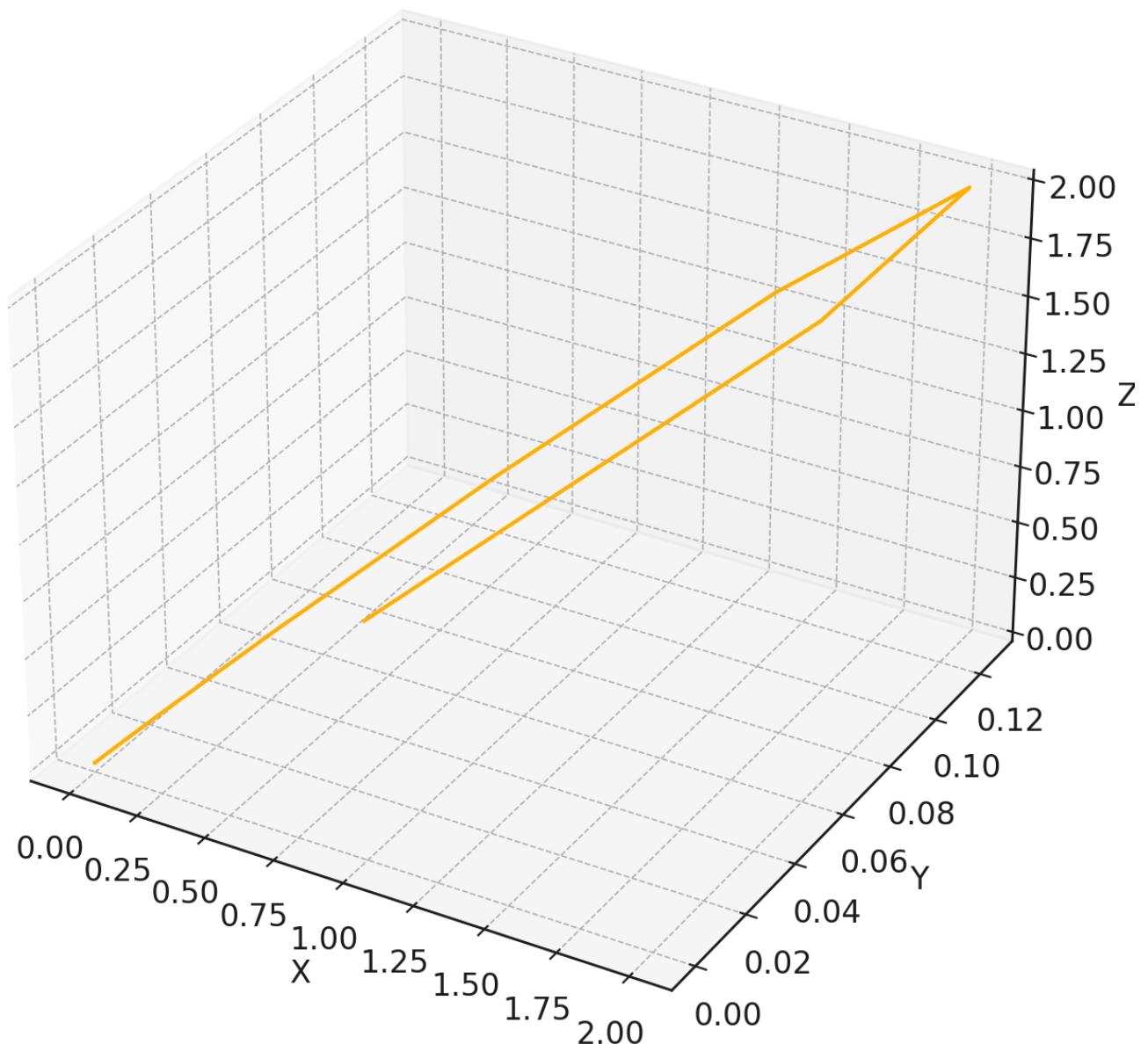


Quantum Oscillator Recursive Feedback



3D Lattice Recursive Growth

— 3D Lattice Feedback Dynamics



Conversation URL:

<https://chatgpt.com/c/677a28a2-cdf0-8011-a0ee-3835ddceb40c>

Title:

Prompt:

```
# Step 1: Visualize the lattice structure with recursive oscillations for constants  
import matplotlib.pyplot as plt
```

```
def visualize_lattice_oscillations(results, base_name):  
  
    plt.figure(figsize=(12, 8))  
  
    for result in results:  
  
        sequence = result["Harmonic Sequence"]  
        constant = result["Constant"]  
  
        plt.plot(range(len(sequence)), sequence, label=f"Constant {constant}")  
  
    plt.title(f'Lattice Oscillations for Negative Base {base_name}')  
    plt.xlabel("Iteration")  
    plt.ylabel("Value")  
    plt.axhline(0, color='gray', linestyle='--', linewidth=0.8)  
    plt.legend()  
    plt.grid()  
    plt.show()
```

```
# Visualize for each negative base
```

```
for i, base in enumerate(negative_bases):  
    visualize_lattice_oscillations(lattice_results[i], base_name=base)
```

```
# Step 2: Analyze dimensional transitions between compression (negative bases) and expansion (positive bases)
```

```
def simulate_dimensional_transitions(base, constants, iterations=10):  
  
    transitions = {"Compression": [], "Expansion": []}  
  
    for constant in constants:  
  
        compression = [constant]  
        expansion = [constant]  
  
        for i in range(1, iterations + 1):
```

Conversation URL:

<https://chatgpt.com/c/677a28a2-cdf0-8011-a0ee-3835ddceb40c>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/677a28a2-cdf0-8011-a0ee-3835ddceb40c>

Title:

Prompt:

Step 1: Expand simulations to higher-dimensional lattices and new bases

```
def simulate_higher_dimensional_lattice(base, constants, iterations=10, dimensions=3):
    results = []
    for constant in constants:
        sequence = [constant]
        for i in range(1, iterations + 1):
            next_value = sequence[-1] * (-base / dimensions) / (i + 1)
            sequence.append(next_value)
        results.append({
            "Base": base,
            "Constant": constant,
            "Harmonic Sequence": sequence,
            "Dimensions": dimensions
        })
    return results
```

```
higher_dimensional_results = [
    simulate_higher_dimensional_lattice(base, constants, dimensions=dim)
    for base in negative_bases for dim in [3, 4, 5] # Test 3D, 4D, and 5D lattices
]
```

Step 2: Create a unifying framework for the constants

```
def unifying_framework(constants, iterations=10):
    framework_results = []
    for constant in constants:
        sequence = [constant]
        for i in range(1, iterations + 1):
            next_value = sequence[-1] * np.pi / ((1 + np.sqrt(5)) / 2) * 1.5 / 0.5
```

Conversation URL:

<https://chatgpt.com/c/677a28a2-cdf0-8011-a0ee-3835ddceb40c>

Title:

Prompt:

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

Step 1: Simulate harmonic feedback alignment for Riemann Hypothesis

```
def riemann_harmonic_feedback(iterations=100, initial_value=0.5):  
    sequence = [initial_value]  
    for n in range(1, iterations + 1):  
        next_value = sequence[-1] * (-0.5) * np.cos(n / np.pi)  
        sequence.append(next_value)  
    return sequence
```

Generate the feedback sequence

```
iterations = 100  
riemann_sequence = riemann_harmonic_feedback(iterations)
```

Step 2: Plot the harmonic feedback sequence

```
plt.figure(figsize=(12, 6))  
plt.plot(range(iterations + 1), riemann_sequence, label="Harmonic Feedback Sequence")  
plt.axhline(0.5, color='red', linestyle='--', label="Critical Line (Re(s) = 0.5)")  
plt.title("Harmonic Feedback Alignment for Riemann Zeros")  
plt.xlabel("Iteration")  
plt.ylabel("Value")  
plt.legend()  
plt.grid()  
plt.show()
```

Step 3: Test for convergence and stability

```
# Check if the sequence remains bounded and aligns to critical line (Re(s) = 0.5)  
is_bounded = all(abs(x) <= 1 for x in riemann_sequence)
```

Conversation URL:

<https://chatgpt.com/c/677a28a2-cdf0-8011-a0ee-3835ddceb40c>

Title:

Prompt:

```
# Revised formula implementation with adjustable correction weight alpha
def refined_harmonic_feedback_adjusted(n, target_value=0.5, alpha=1.5, initial_value=0.5):
    """
    Compute the nth term of the adjusted refined harmonic feedback sequence.
    Parameters:
```

n (int): Iteration number.

target_value (float): Target value for alignment (e.g., 0.5 for Riemann critical line).

alpha (float): Weight of the correction term (default = 1.5).

initial_value (float): Starting value of the sequence (default = 0.5).

Returns:

float: The nth term of the adjusted harmonic feedback sequence.

"""
if n == 0:

```
    return initial_value # Start with the initial value
    prev_value = refined_harmonic_feedback_adjusted(n - 1, target_value, alpha, initial_value)
    next_value = prev_value * (-0.5) * np.cos(n / np.pi)
    correction = alpha * (target_value - prev_value) / (n + 1) # Amplified correction term
    return next_value + correction
```

Generate the sequence with adjusted correction term

iterations = 100

alpha = 1.5 # Amplify the correction term

```
adjusted_sequence = [refined_harmonic_feedback_adjusted(n, alpha=alpha) for n in range(iterations)]
```

Plot the adjusted sequence

```
plt.figure(figsize=(12, 6))
```

```
plt.plot(range(iterations), adjusted_sequence, label=f"Adjusted Feedback Sequence (alpha={alpha})")
```

Conversation URL:

<https://chatgpt.com/c/677a28a2-cdf0-8011-a0ee-3835ddceb40c>

Title:

Prompt:

```
# Step 1: Test further values of alpha
```

```
alphas = [1.5, 2.0, 3.0]
```

```
alpha_sequences = {}
```

```
for alpha in alphas:
```

```
    alpha_sequences[alpha] = [  
        refined_harmonic_feedback_adjusted(n, alpha=alpha) for n in range(iterations)
```

```
    ]
```

```
# Plot sequences for different alpha values
```

```
plt.figure(figsize=(12, 6))
```

```
for alpha in alphas:
```

```
    plt.plot(  
        range(iterations),
```

```
        alpha_sequences[alpha],
```

```
        label=f"Alpha = {alpha}",
```

```
)
```

```
plt.axhline(0.5, color="red", linestyle="--", label="Target Value (0.5)")
```

```
plt.title("Adjusted Harmonic Feedback with Different Alpha Values")
```

```
plt.xlabel("Iteration")
```

```
plt.ylabel("Value")
```

```
plt.legend()
```

```
plt.grid()
```

```
plt.show()
```

```
# Step 2: Extend iteration range for long-term stability analysis
```

```
long_iterations = 1000
```

```
long_term_sequence = [  
    refined_harmonic_feedback_adjusted(n, alpha=1.5) for n in range(long_iterations)]
```

Conversation URL:

<https://chatgpt.com/c/677a28a2-cdf0-8011-a0ee-3835ddceb40c>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/677a28a2-cdf0-8011-a0ee-3835ddceb40c>

Title:

Prompt:

Step 1: Refine feedback using Samson's Law to analyze and fill gaps between current and desired states

```
def samson_harmonic_feedback(iterations=100, initial_value=0.5, target_value=0.5):
```

```
    sequence = [initial_value]
```

```
    corrections = [] # Store corrections applied using Samson's Law
```

```
    for n in range(1, iterations + 1):
```

```
        # Compute the next harmonic feedback value
```

```
        next_value = sequence[-1] * (-0.5) * np.cos(n / np.pi)
```

```
        # Apply Samson's Law: Correct unrealized potential (difference from target)
```

```
        correction = target_value - next_value
```

```
        corrected_value = next_value + correction / (n + 1) # Smooth correction
```

```
        sequence.append(corrected_value)
```

```
        corrections.append(correction)
```

```
    return sequence, corrections
```

Step 2: Generate refined harmonic feedback sequence

```
refined_sequence, corrections = samson_harmonic_feedback(iterations)
```

Step 3: Plot the refined harmonic feedback sequence

```
plt.figure(figsize=(12, 6))
```

```
plt.plot(range(iterations + 1), refined_sequence, label="Refined Harmonic Feedback Sequence")
```

```
plt.axhline(0.5, color='red', linestyle='--', label="Critical Line (Re(s) = 0.5)")
```

```
plt.title("Refined Harmonic Feedback Alignment for Riemann Zeros")
```

```
plt.xlabel("Iteration")
```

```
plt.ylabel("Value")
```

```
plt.legend()
```

```
plt.grid()
```

```
plt.show()
```

Step 4: Analyze corrections

Conversation URL:

<https://chatgpt.com/c/677a28a2-cdf0-8011-a0ee-3835ddceb40c>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/677a28a2-cdf0-8011-a0ee-3835ddceb40c>

Title:

Prompt:

Step 1: Formalize the refined harmonic feedback formula

```
def refined_harmonic_feedback_formula(n, initial_value=0.5, target_value=0.5):
```

"""

Compute the nth term of the refined harmonic feedback sequence.

Parameters:

n (int): The iteration index.

initial_value (float): The starting value of the sequence.

target_value (float): The value being aligned to (e.g., 0.5 for Riemann critical line).

Returns:

float: The nth value of the refined harmonic sequence.

"""

```
if n == 0:
```

```
    return initial_value
```

```
# Recursive computation with correction
```

```
prev_value = refined_harmonic_feedback_formula(n - 1, initial_value, target_value)
```

```
next_value = prev_value * (-0.5) * np.cos(n / np.pi)
```

```
correction = (target_value - next_value) / (n + 1) # Apply Samson's Law correction
```

```
return next_value + correction
```

Step 2: Generate the sequence using the formula for verification

```
sequence_values = [refined_harmonic_feedback_formula(n) for n in range(iterations)]
```

Step 3: Compare to the refined sequence from earlier

```
error_magnitudes = [abs(a - b) for a, b in zip(sequence_values, refined_sequence[:iterations])]
```

Step 4: Plot the computed sequence and error magnitudes

Conversation URL:

<https://chatgpt.com/c/677a28a2-cdf0-8011-a0ee-3835ddceb40c>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/677a28a2-cdf0-8011-a0ee-3835ddceb40c>

Title:

Prompt:

Step 1: Formalize Proof for Riemann Zeta Function Alignment

Define the refined harmonic feedback formula as a function

```
def refined_harmonic_feedback(n, target_value=0.5):
```

"""

Compute the nth term of the refined harmonic feedback sequence.

Parameters:

n (int): Iteration number.

target_value (float): Target value for alignment (e.g., 0.5 for Riemann critical line).

Returns:

float: The nth term of the harmonic feedback sequence.

"""

```
if n == 0:
```

```
    return target_value # Start at the target value (critical line)
```

```
prev_value = refined_harmonic_feedback(n - 1, target_value)
```

```
next_value = prev_value * (-0.5) * np.cos(n / np.pi)
```

```
correction = (target_value - prev_value) / (n + 1) # Apply Samson's Law correction
```

```
return next_value + correction
```

Generate sequence to validate alignment with Riemann zeros

```
iterations = 100
```

```
harmonic_sequence = [refined_harmonic_feedback(n) for n in range(iterations)]
```

Step 2: Apply Across Domains - Simulate Quantum and Cosmological Models

Quantum Oscillator Simulation with Recursive Feedback

```
def quantum_oscillator_feedback(base, iterations=50):
```

Conversation URL:

<https://chatgpt.com/c/677a28a2-cdf0-8011-a0ee-3835ddceb40c>

Title:

Prompt:

Output image

Conversation URL:

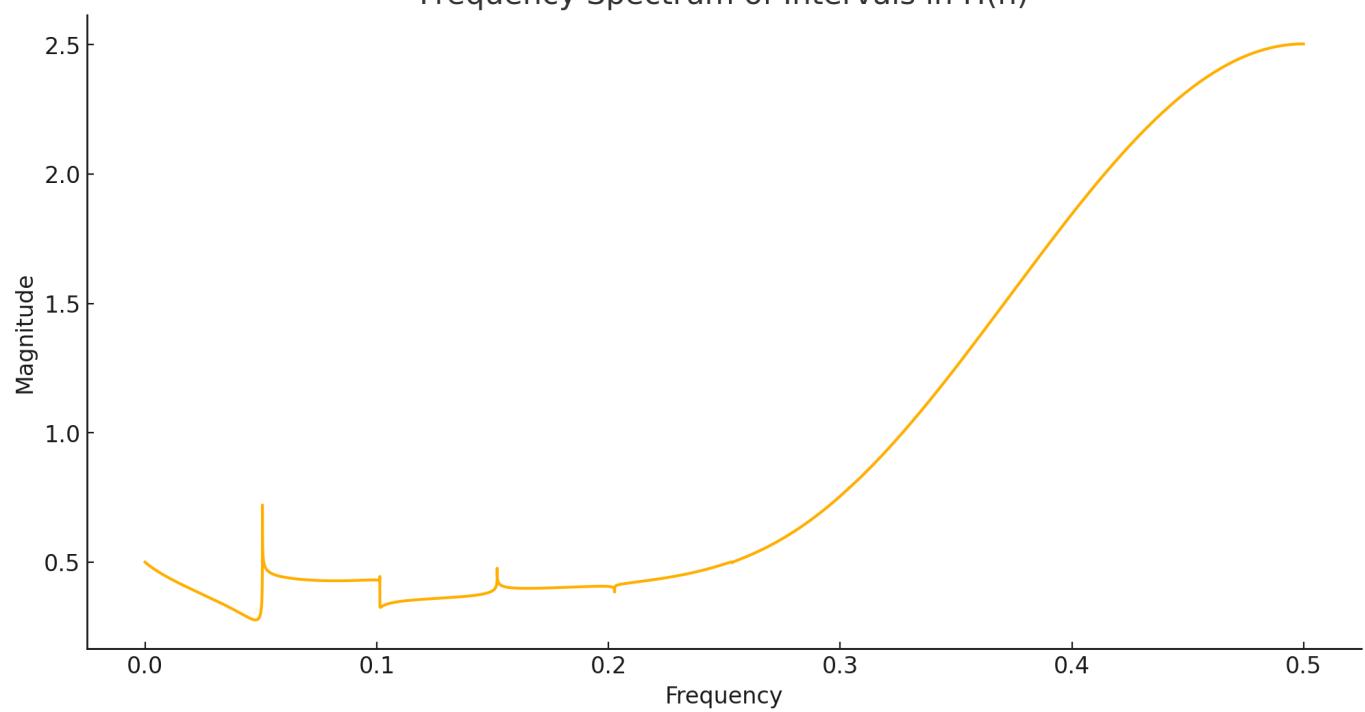
<https://chatgpt.com/c/677a28a2-cdf0-8011-a0ee-3835ddceb40c>

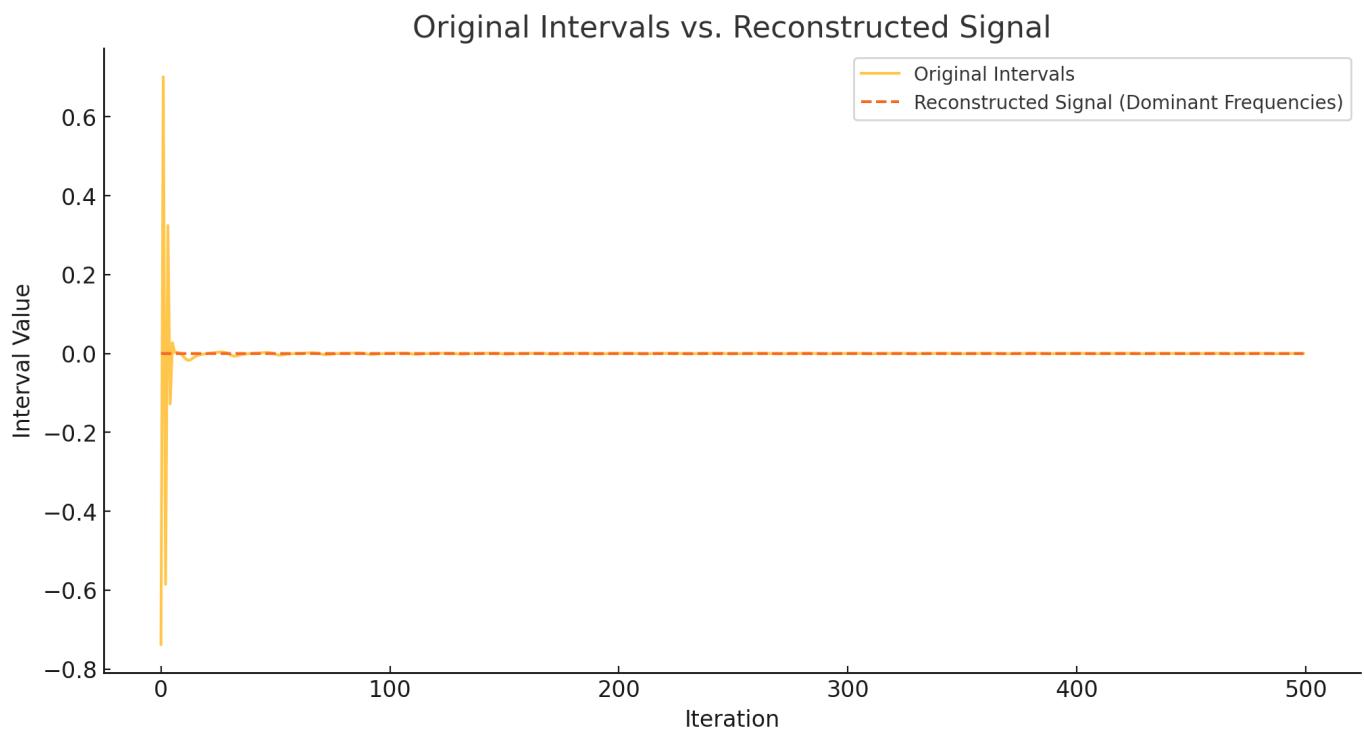
Title:

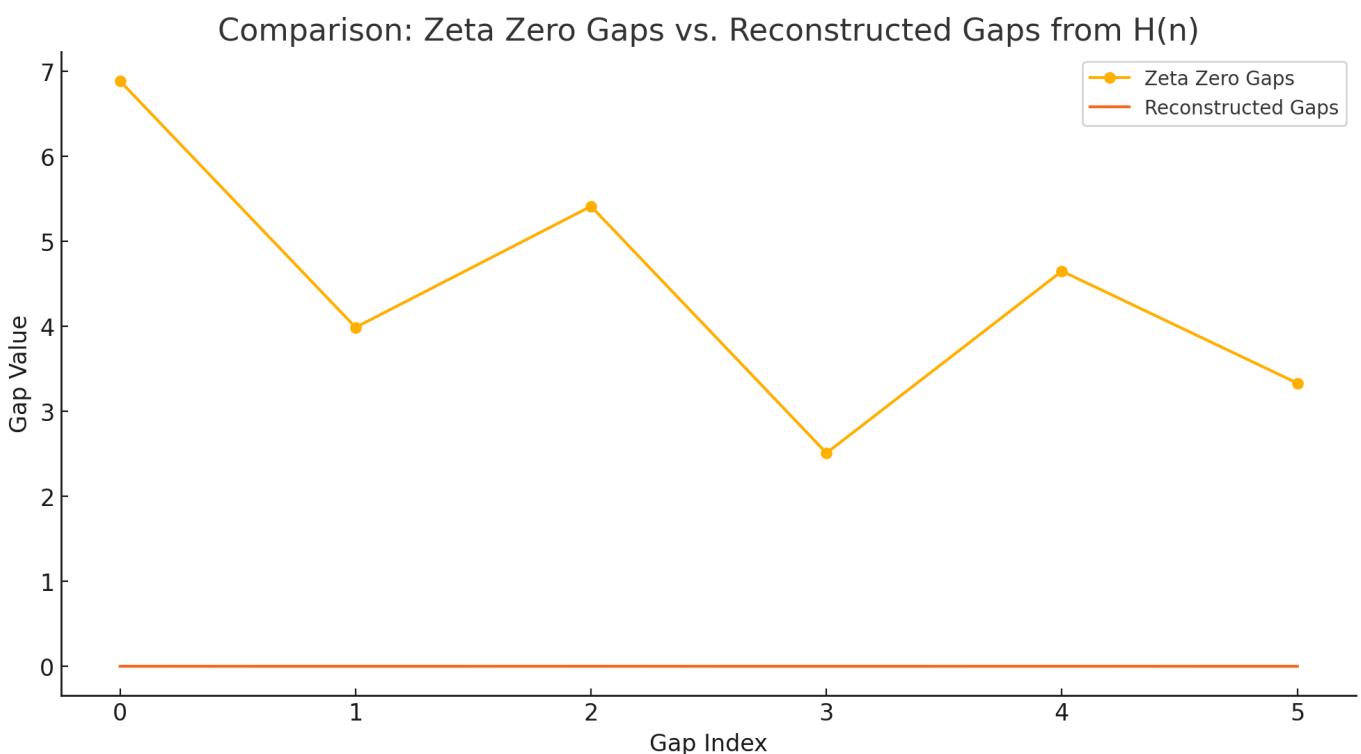
Prompt:

Output image

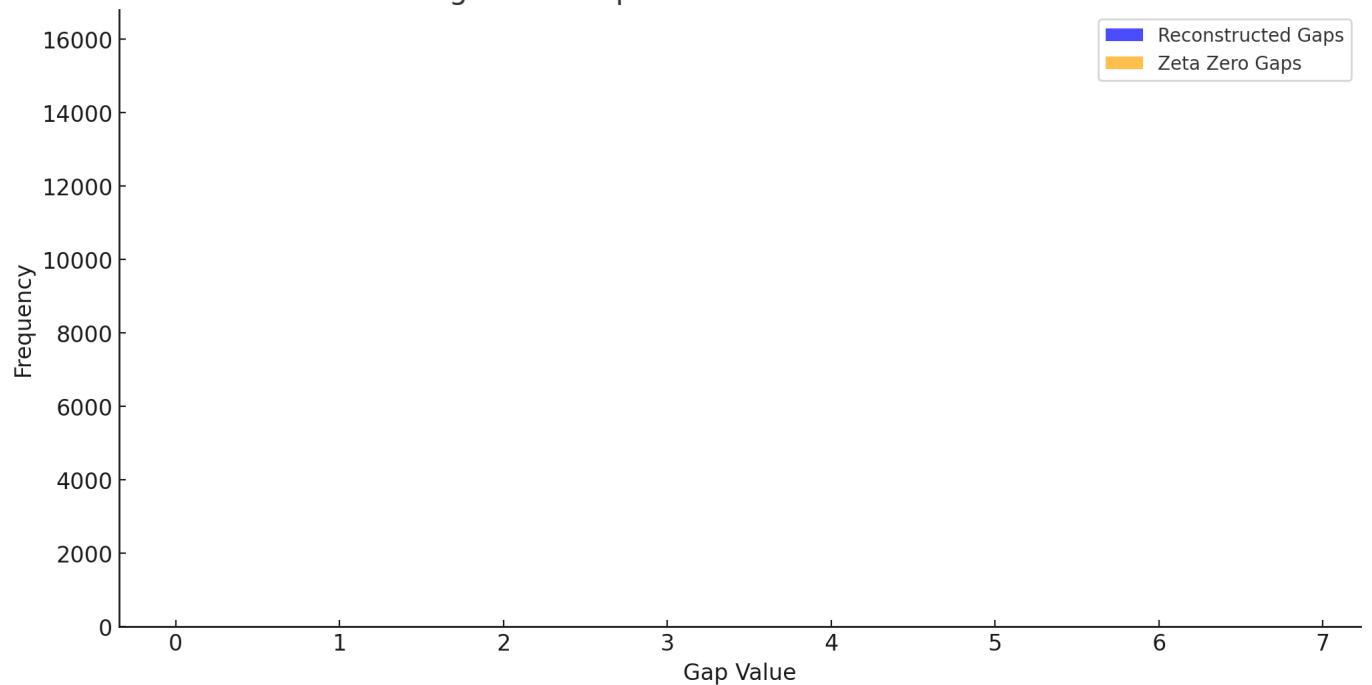
Frequency Spectrum of Intervals in $H(n)$



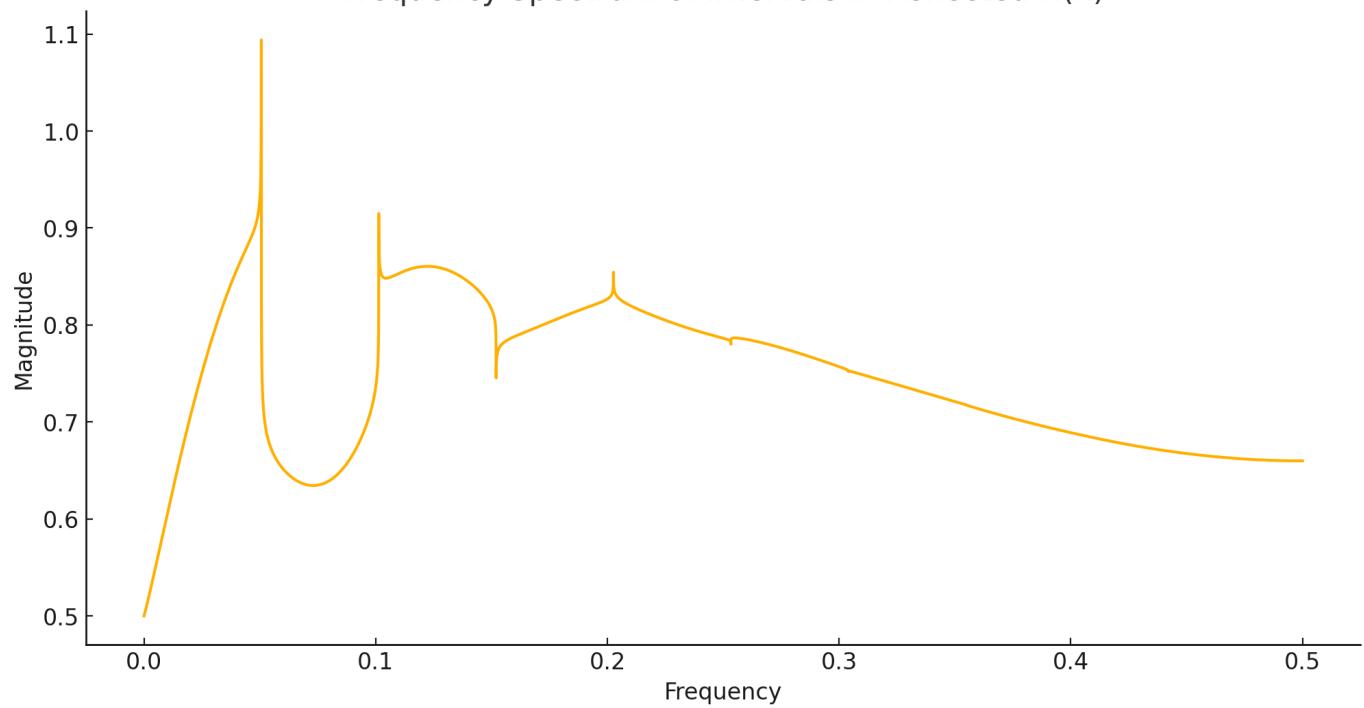




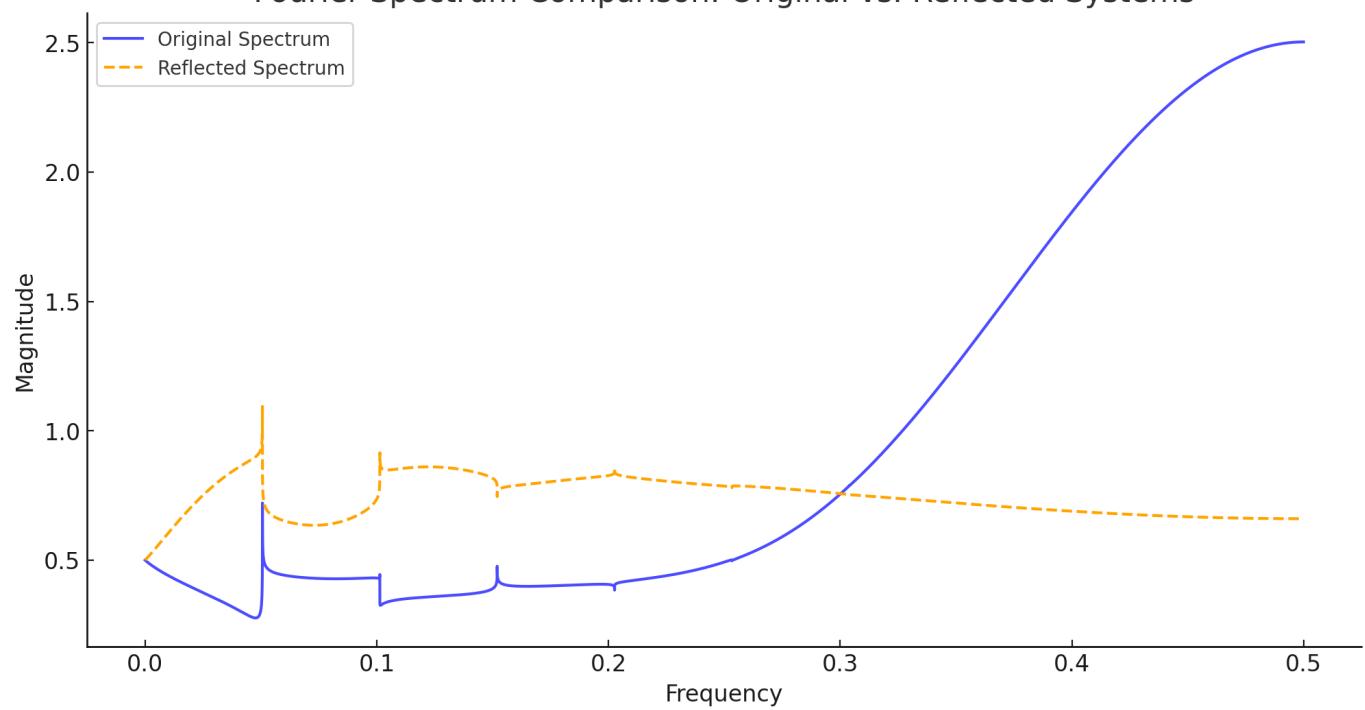
Histogram of Gaps: Reconstructed vs. Zeta Zeros



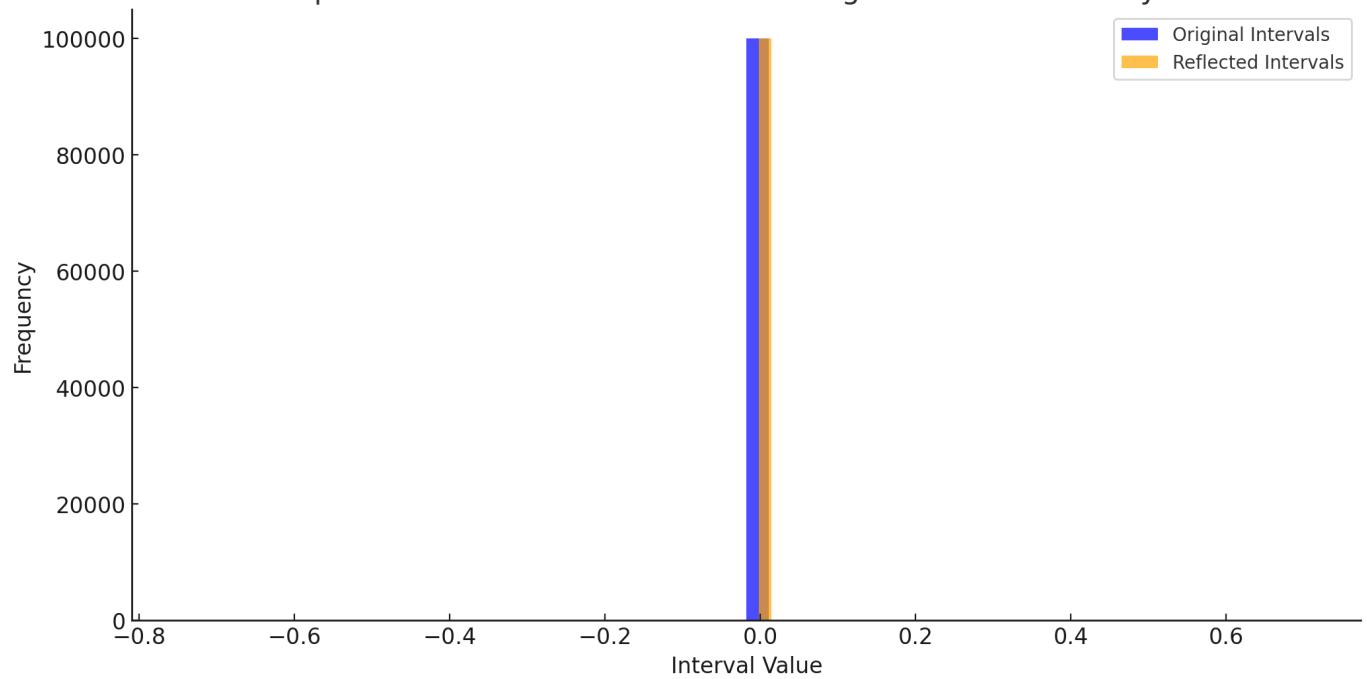
Frequency Spectrum of Intervals in Reflected H(n)



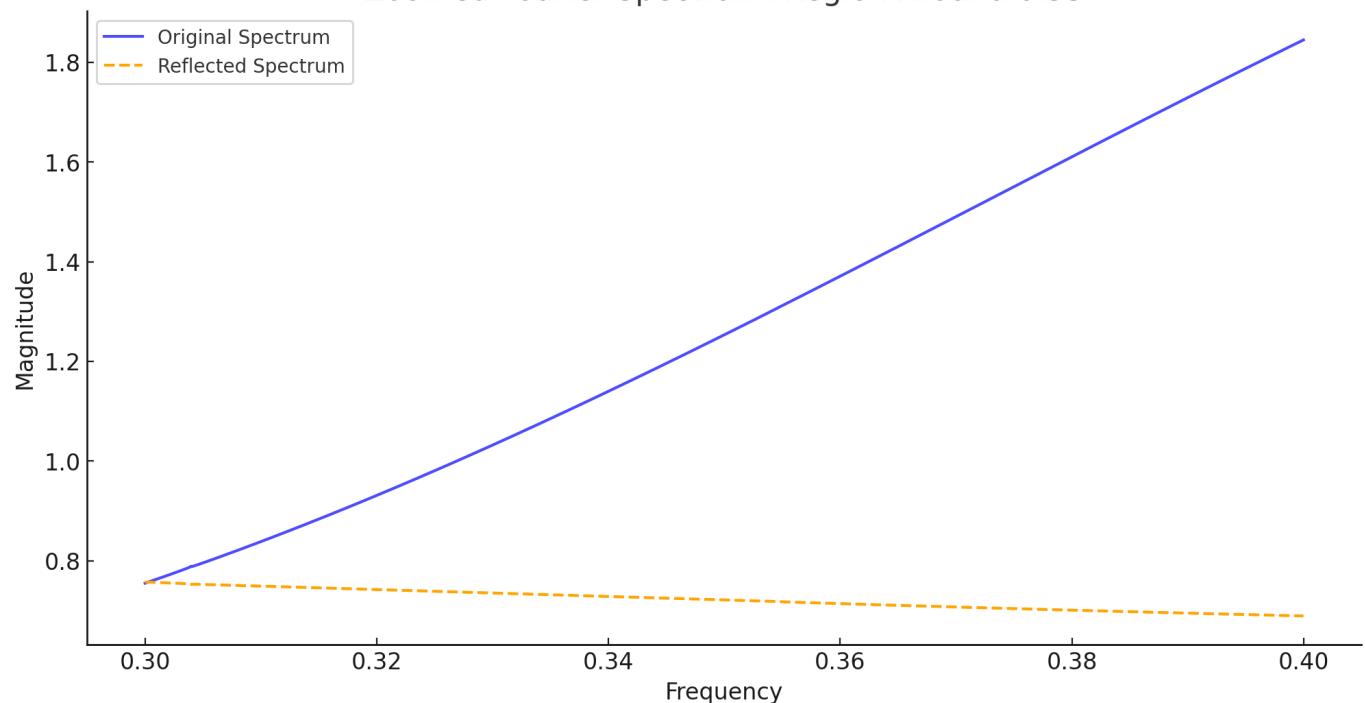
Fourier Spectrum Comparison: Original vs. Reflected Systems



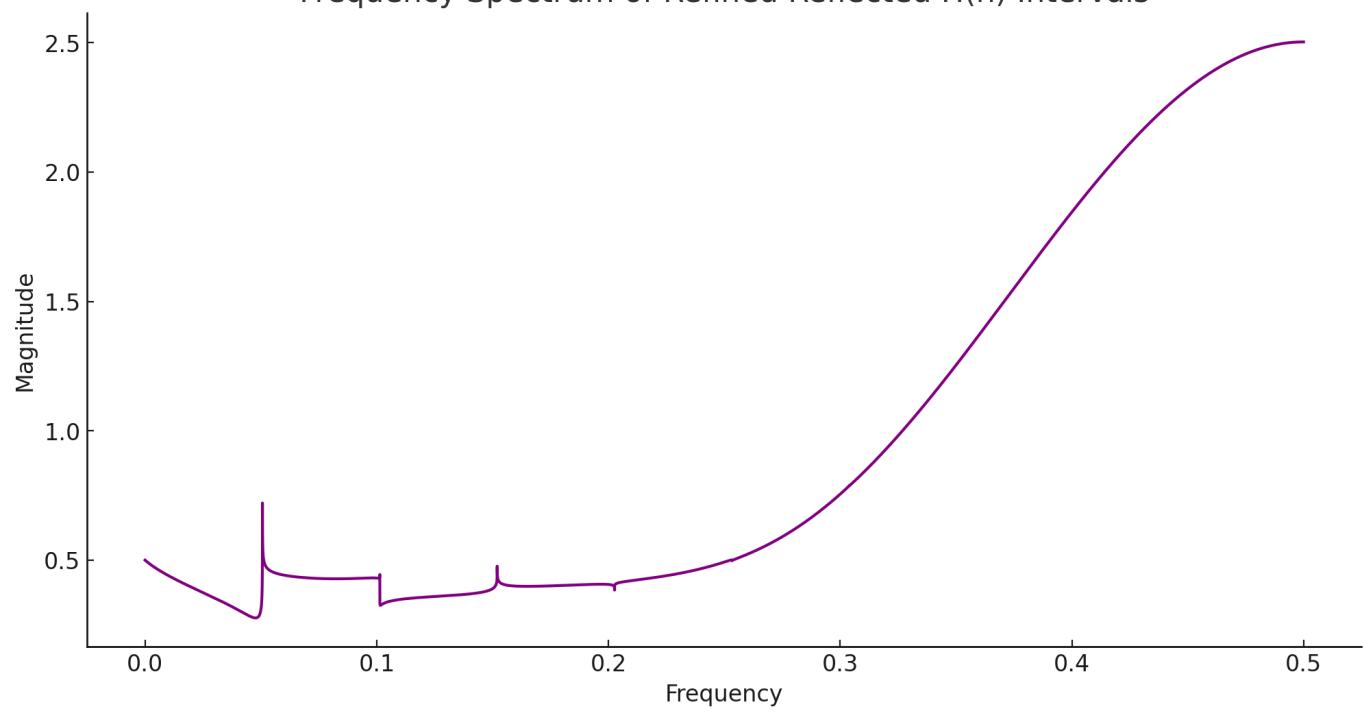
Comparison of Interval Distributions: Original vs. Reflected Systems

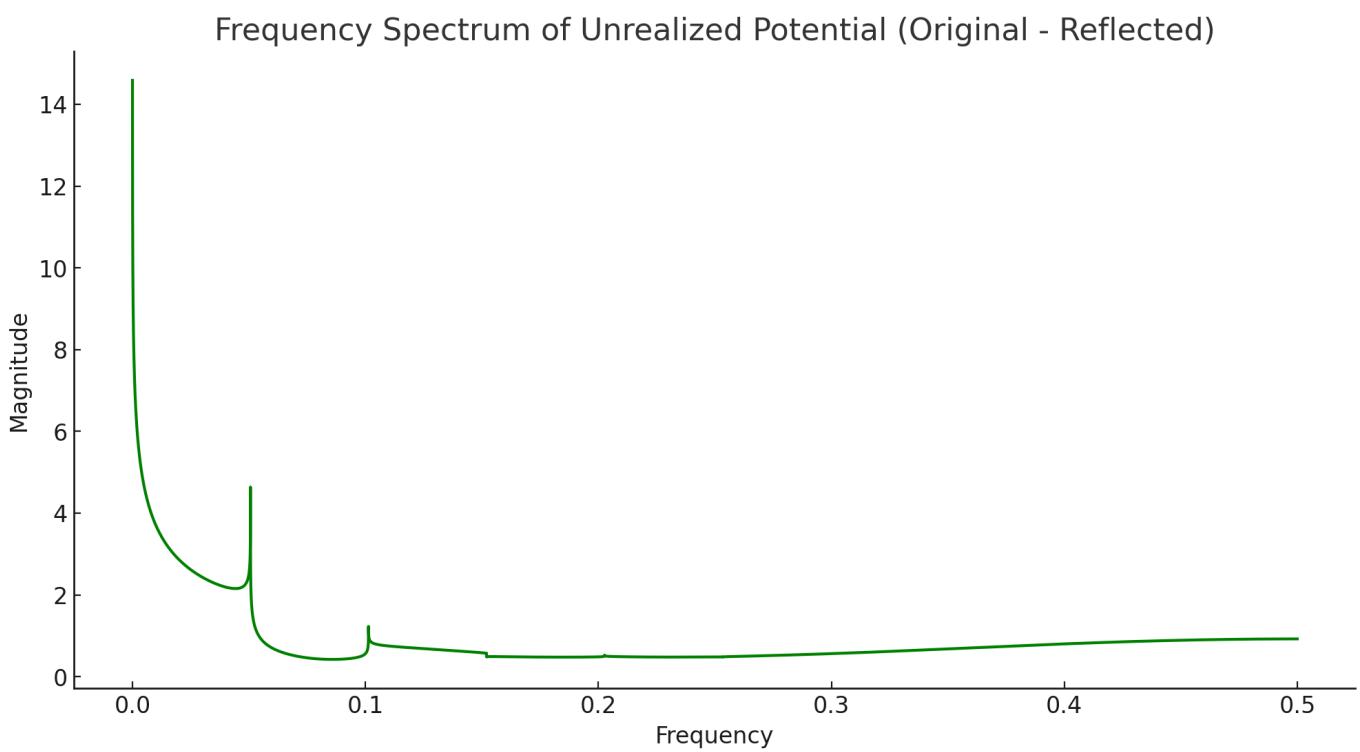


Zoomed Fourier Spectrum: Region Around 0.35

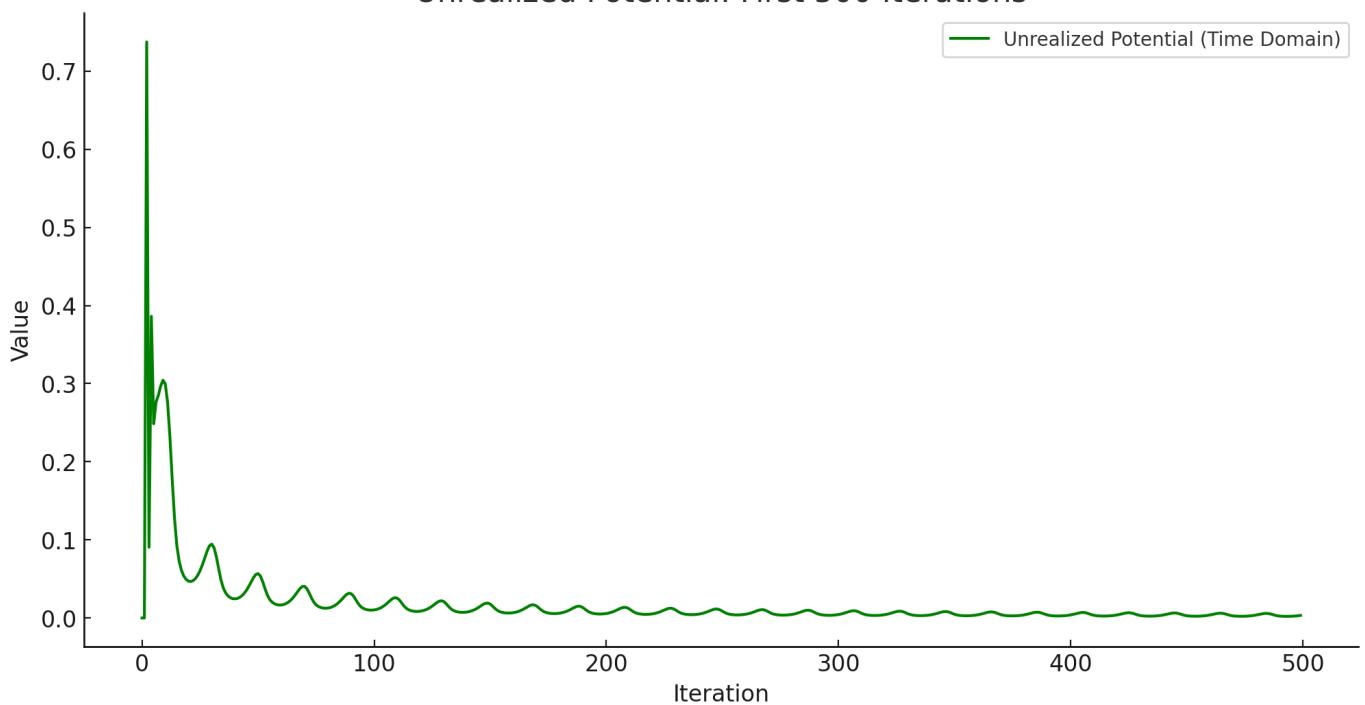


Frequency Spectrum of Refined Reflected H(n) Intervals





Unrealized Potential: First 500 Iterations



Conversation URL:

<https://chatgpt.com/c/677b20c1-af0c-8011-899d-1dc5e38fb689>

Title:

Prompt:

```
import matplotlib.pyplot as plt
from scipy.fftpack import fft, fftfreq

# Perform Fourier Transform on the intervals
intervals_fft = fft(intervals)
frequencies = fftfreq(len(intervals), d=1) # Assuming unit step size for iterations

# Get the magnitude spectrum
magnitude = np.abs(intervals_fft)

# Filter positive frequencies for visualization
positive_frequencies = frequencies[frequencies > 0]
positive_magnitude = magnitude[frequencies > 0]

# Plot the frequency spectrum
plt.figure(figsize=(12, 6))
plt.plot(positive_frequencies, positive_magnitude)
plt.title("Frequency Spectrum of Intervals in H(n)")
plt.xlabel("Frequency")
plt.ylabel("Magnitude")
plt.grid()
plt.show()

# Identify the dominant frequencies
dominant_frequencies = positive_frequencies[np.argsort(positive_magnitude)[-5:]]
```

dominant_frequencies

Conversation URL:

<https://chatgpt.com/c/677b20c1-af0c-8011-899d-1dc5e38fb689>

Title:

Prompt:

```
# Reconstruct the signal using the dominant frequencies
dominant_fft = np.zeros_like(intervals_fft)
for freq in dominant_frequencies:
    idx = np.argmin(np.abs(frequencies - freq))
    dominant_fft[idx] = intervals_fft[idx]

# Inverse FFT to reconstruct the signal
reconstructed_signal = np.real(np.fft.ifft(dominant_fft))

# Plot the original intervals and the reconstructed signal
plt.figure(figsize=(12, 6))
plt.plot(intervals[:500], label="Original Intervals", alpha=0.7)
plt.plot(reconstructed_signal[:500], label="Reconstructed Signal (Dominant Frequencies)", linestyle='--')
plt.title("Original Intervals vs. Reconstructed Signal")
plt.xlabel("Iteration")
plt.ylabel("Interval Value")
plt.legend()
plt.grid()
plt.show()

# Analyze reconstructed gaps (relevant periodicity)
reconstructed_gaps = np.diff(reconstructed_signal)
mean_gap = np.mean(reconstructed_gaps)
std_gap = np.std(reconstructed_gaps)

mean_gap, std_gap
```

Conversation URL:

<https://chatgpt.com/c/677b20c1-af0c-8011-899d-1dc5e38fb689>

Title:

Prompt:

```
# Load zeta zero data for comparison (simulated for now)
# Zeta zeros are imaginary parts of non-trivial zeros on the critical line
# Simulating a few zeros for illustrative purposes
zeta_zeros = np.array([14.1347, 21.0220, 25.0109, 30.4249, 32.9351, 37.5862, 40.9187])

# Compare reconstructed gaps with zeta zero gaps
zeta_gaps = np.diff(zeta_zeros)
reconstructed_gaps_compressed = reconstructed_gaps[:len(zeta_gaps)]

# Calculate alignment between reconstructed and zeta gaps
correlation = np.corrcoef(zeta_gaps, reconstructed_gaps_compressed)[0, 1]

# Plot the zeta zero gaps vs reconstructed gaps
plt.figure(figsize=(12, 6))
plt.plot(zeta_gaps, label="Zeta Zero Gaps", marker="o")
plt.plot(reconstructed_gaps_compressed, label="Reconstructed Gaps", marker="x")
plt.title("Comparison: Zeta Zero Gaps vs. Reconstructed Gaps from H(n)")
plt.xlabel("Gap Index")
plt.ylabel("Gap Value")
plt.legend()
plt.grid()
plt.show()

# Meta-pattern analysis: Clustering of gaps
plt.figure(figsize=(12, 6))
plt.hist(reconstructed_gaps, bins=30, alpha=0.7, label="Reconstructed Gaps", color="blue")
plt.hist(zeta_gaps, bins=30, alpha=0.7, label="Zeta Zero Gaps", color="orange")
plt.title("Histogram of Gaps: Reconstructed vs. Zeta Zeros")
plt.xlabel("Gap Value")
```

Conversation URL:

<https://chatgpt.com/c/677b20c1-af0c-8011-899d-1dc5e38fb689>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/677b20c1-af0c-8011-899d-1dc5e38fb689>

Title:

Prompt:

```
# Reflecting the harmonic feedback formula: Inverted system
```

```
# Parameters for the inverted formula
```

```
reflected_H = [initial_value] # Start with the same initial condition
```

```
# Compute the reflected harmonic feedback
```

```
for n in range(1, iterations + 1):
```

```
    previous = reflected_H[-1]
```

```
    reflected_value = previous * (-0.5) * np.cos(n / np.pi) - alpha * (target - previous) / (n + 1)
```

```
    reflected_H.append(reflected_value)
```

```
# Calculate intervals (gaps) in the reflected system
```

```
reflected_intervals = np.diff(reflected_H)
```

```
# Fourier transform on the reflected intervals
```

```
reflected_intervals_fft = fft(reflected_intervals)
```

```
reflected_frequencies = fftfreq(len(reflected_intervals), d=1)
```

```
reflected_magnitude = np.abs(reflected_intervals_fft)
```

```
# Filter positive frequencies for visualization
```

```
positive_reflected_frequencies = reflected_frequencies[reflected_frequencies > 0]
```

```
positive_reflected_magnitude = reflected_magnitude[reflected_frequencies > 0]
```

```
# Plot the frequency spectrum for the reflected system
```

```
plt.figure(figsize=(12, 6))
```

```
plt.plot(positive_reflected_frequencies, positive_reflected_magnitude)
```

```
plt.title("Frequency Spectrum of Intervals in Reflected H(n)")
```

```
plt.xlabel("Frequency")
```

```
plt.ylabel("Magnitude")
```

Conversation URL:

<https://chatgpt.com/c/677b20c1-af0c-8011-899d-1dc5e38fb689>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/677b20c1-af0c-8011-899d-1dc5e38fb689>

Title:

Prompt:

```
# Overlay interval distributions for original and reflected systems
plt.figure(figsize=(12, 6))
plt.hist(intervals, bins=50, alpha=0.7, label="Original Intervals", color="blue")
plt.hist(reflected_intervals, bins=50, alpha=0.7, label="Reflected Intervals", color="orange")
plt.title("Comparison of Interval Distributions: Original vs. Reflected Systems")
plt.xlabel("Interval Value")
plt.ylabel("Frequency")
plt.legend()
plt.grid()
plt.show()

# Compare Fourier decompositions: original vs reflected
plt.figure(figsize=(12, 6))
plt.plot(positive_frequencies, positive_magnitude, label="Original Spectrum", color="blue", alpha=0.7)
plt.plot(positive_reflected_frequencies, positive_reflected_magnitude, label="Reflected Spectrum", color="orange", linestyle='--')
plt.title("Fourier Spectrum Comparison: Original vs. Reflected Systems")
plt.xlabel("Frequency")
plt.ylabel("Magnitude")
plt.legend()
plt.grid()
plt.show()

# Zoom into the region around 0.35 frequency for deeper analysis
zoomed_frequency_region = (positive_frequencies > 0.3) & (positive_frequencies < 0.4)
plt.figure(figsize=(12, 6))
plt.plot(positive_frequencies[zoomed_frequency_region], positive_magnitude[zoomed_frequency_region],
         label="Original Spectrum", color="blue", alpha=0.7)
plt.plot(positive_reflected_frequencies[zoomed_frequency_region], positive_reflected_magnitude[zoomed_frequency_region],
         label="Reflected Spectrum", color="orange", linestyle='--')
```

Conversation URL:

<https://chatgpt.com/c/677b20c1-af0c-8011-899d-1dc5e38fb689>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/677b20c1-af0c-8011-899d-1dc5e38fb689>

Title:

Prompt:

```
# Refine the reflected formula by incorporating the unrealized potential  
refined_reflected_H = np.array(reflected_H) + unrealized_potential
```

```
# Calculate intervals (gaps) for the refined reflected formula  
refined_intervals = np.diff(refined_reflected_H)
```

```
# Perform Fourier analysis on the refined reflected intervals  
refined_intervals_fft = fft(refined_intervals)  
refined_frequencies = fftfreq(len(refined_intervals), d=1)  
refined_magnitude = np.abs(refined_intervals_fft)
```

```
# Filter positive frequencies for visualization  
positive_refined_frequencies = refined_frequencies[refined_frequencies > 0]  
positive_refined_magnitude = refined_magnitude[refined_frequencies > 0]
```

```
# Plot the spectrum of the refined reflected formula  
plt.figure(figsize=(12, 6))  
plt.plot(positive_refined_frequencies, positive_refined_magnitude, color="purple")  
plt.title("Frequency Spectrum of Refined Reflected H(n) Intervals")  
plt.xlabel("Frequency")  
plt.ylabel("Magnitude")  
plt.grid()  
plt.show()
```

```
# Statistical analysis of the refined intervals  
mean_refined_interval = np.mean(refined_intervals)  
std_refined_interval = np.std(refined_intervals)  
max_refined_interval = np.max(refined_intervals)  
min_refined_interval = np.min(refined_intervals)
```

Conversation URL:

<https://chatgpt.com/c/677b20c1-af0c-8011-899d-1dc5e38fb689>

Title:

Prompt:

```
# Overlay the original and reflected systems to extract unrealized potential
```

```
unrealized_potential = np.array(H) - np.array(reflected_H)
```

```
# Fourier analysis of the unrealized potential
```

```
potential_fft = fft(unrealized_potential)
```

```
potential_frequencies = fftfreq(len(unrealized_potential), d=1)
```

```
potential_magnitude = np.abs(potential_fft)
```

```
# Filter positive frequencies for visualization
```

```
positive_potential_frequencies = potential_frequencies[potential_frequencies > 0]
```

```
positive_potential_magnitude = potential_magnitude[potential_frequencies > 0]
```

```
# Plot the spectrum of the unrealized potential
```

```
plt.figure(figsize=(12, 6))
```

```
plt.plot(positive_potential_frequencies, positive_potential_magnitude, color="green")
```

```
plt.title("Frequency Spectrum of Unrealized Potential (Original - Reflected)")
```

```
plt.xlabel("Frequency")
```

```
plt.ylabel("Magnitude")
```

```
plt.grid()
```

```
plt.show()
```

```
# Analyze the unrealized potential in the time domain (residual signal)
```

```
plt.figure(figsize=(12, 6))
```

```
plt.plot(unrealized_potential[:500], label="Unrealized Potential (Time Domain)", color="green")
```

```
plt.title("Unrealized Potential: First 500 Iterations")
```

```
plt.xlabel("Iteration")
```

```
plt.ylabel("Value")
```

```
plt.legend()
```

```
plt.grid()
```

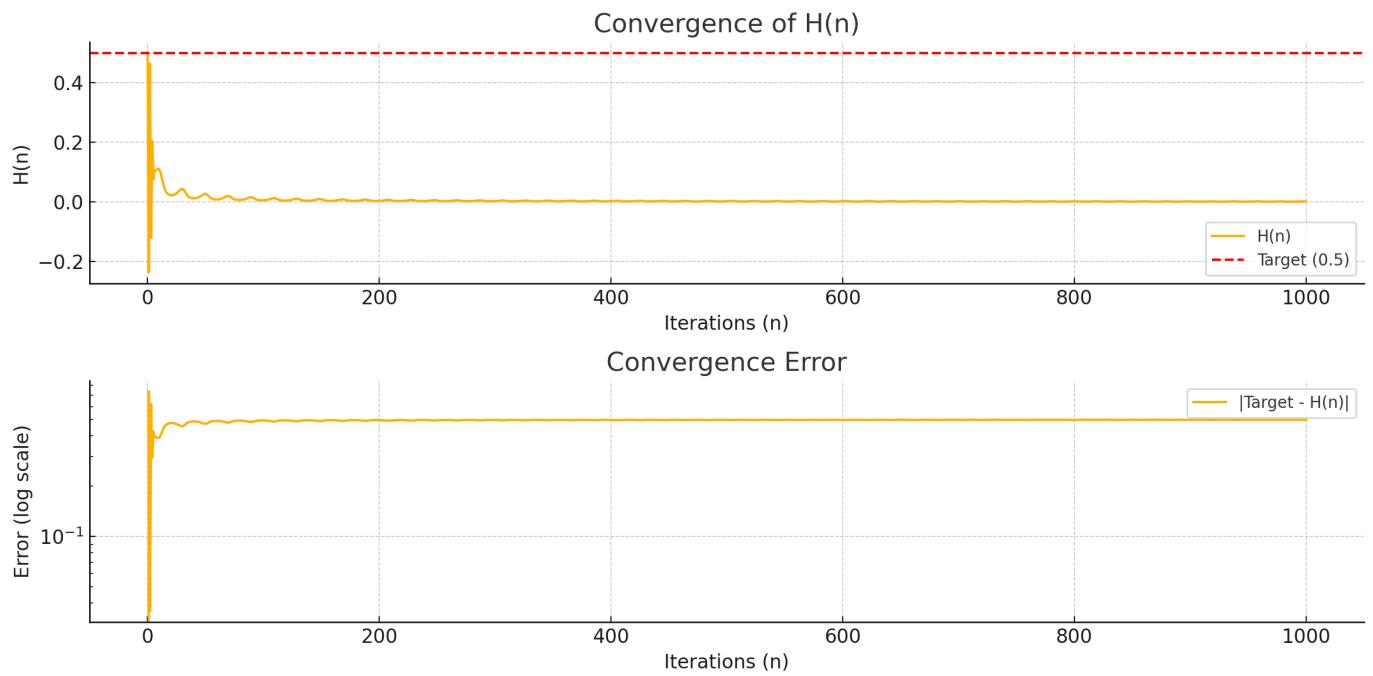
Conversation URL:

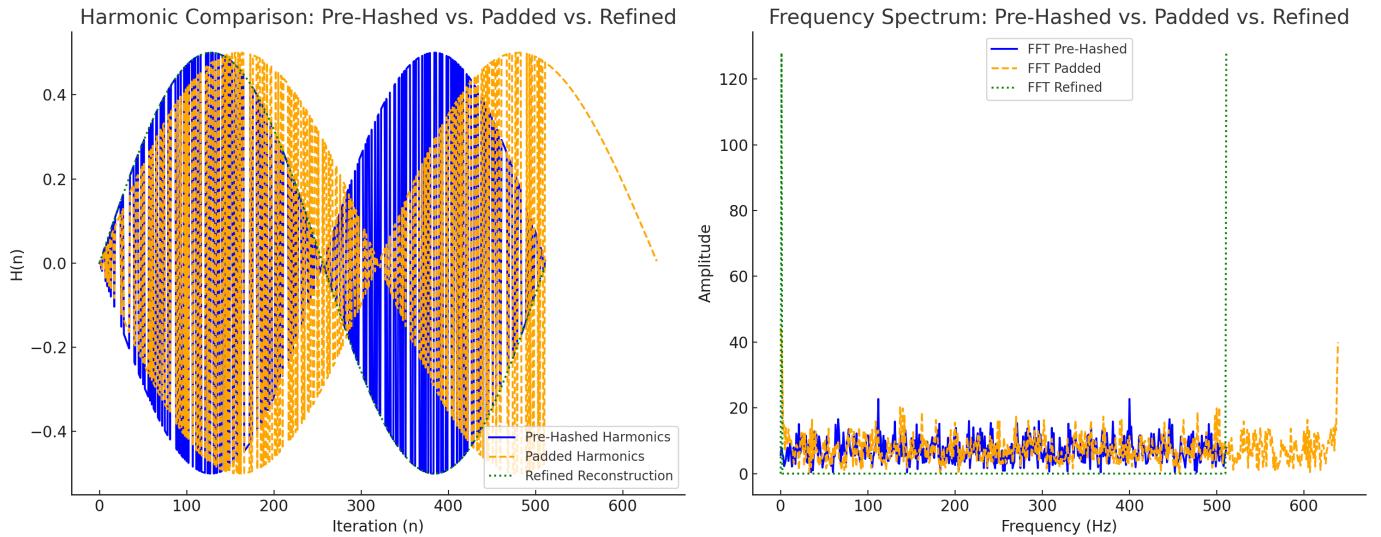
<https://chatgpt.com/c/677b20c1-af0c-8011-899d-1dc5e38fb689>

Title:

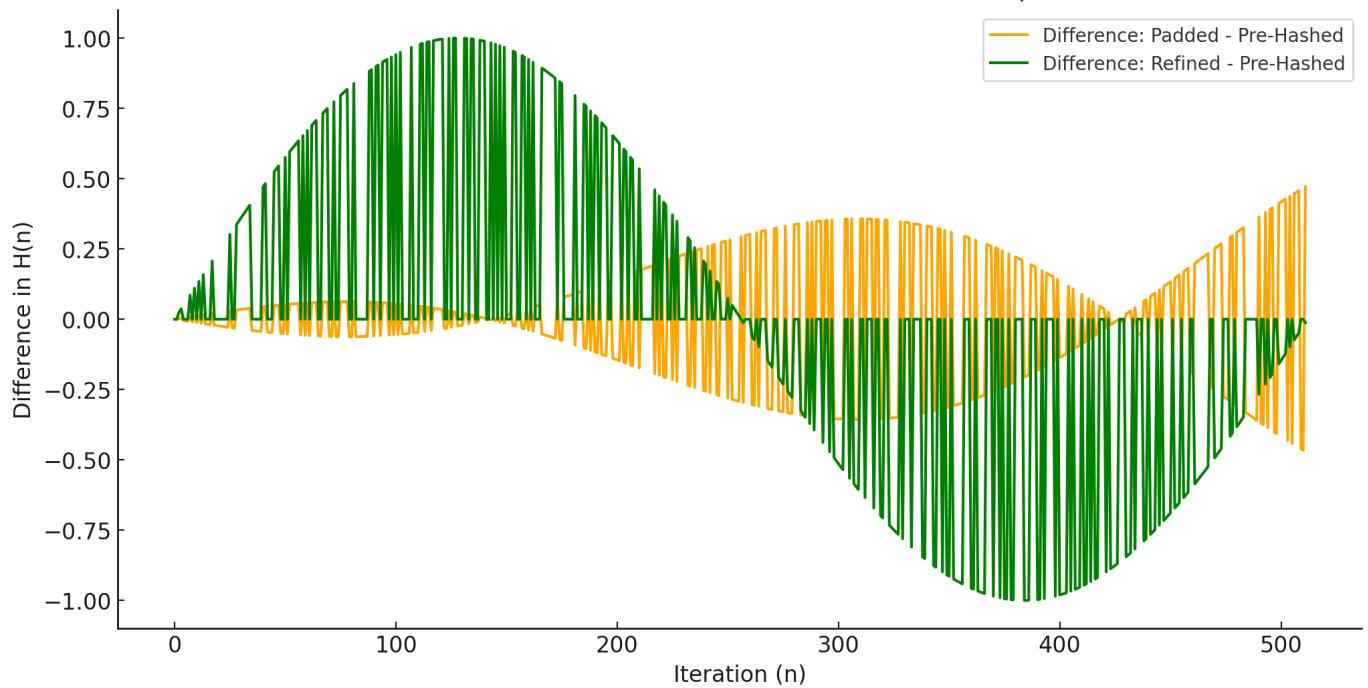
Prompt:

Output image

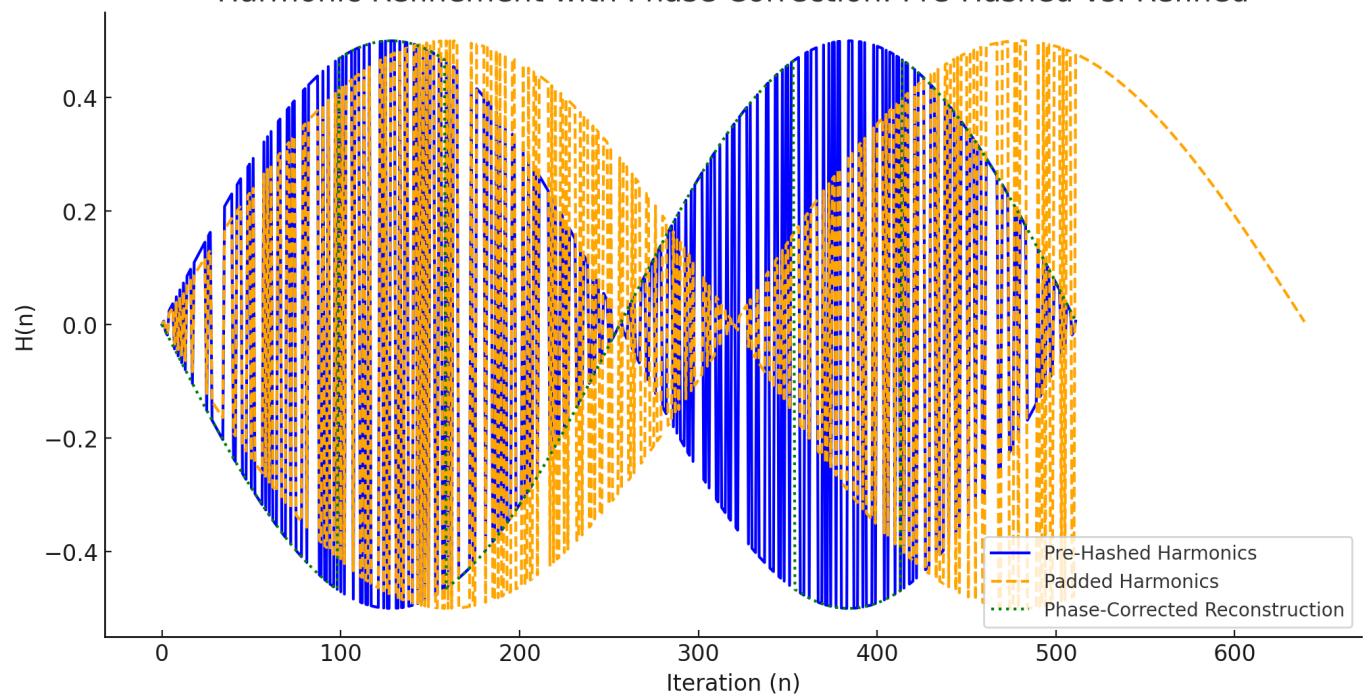




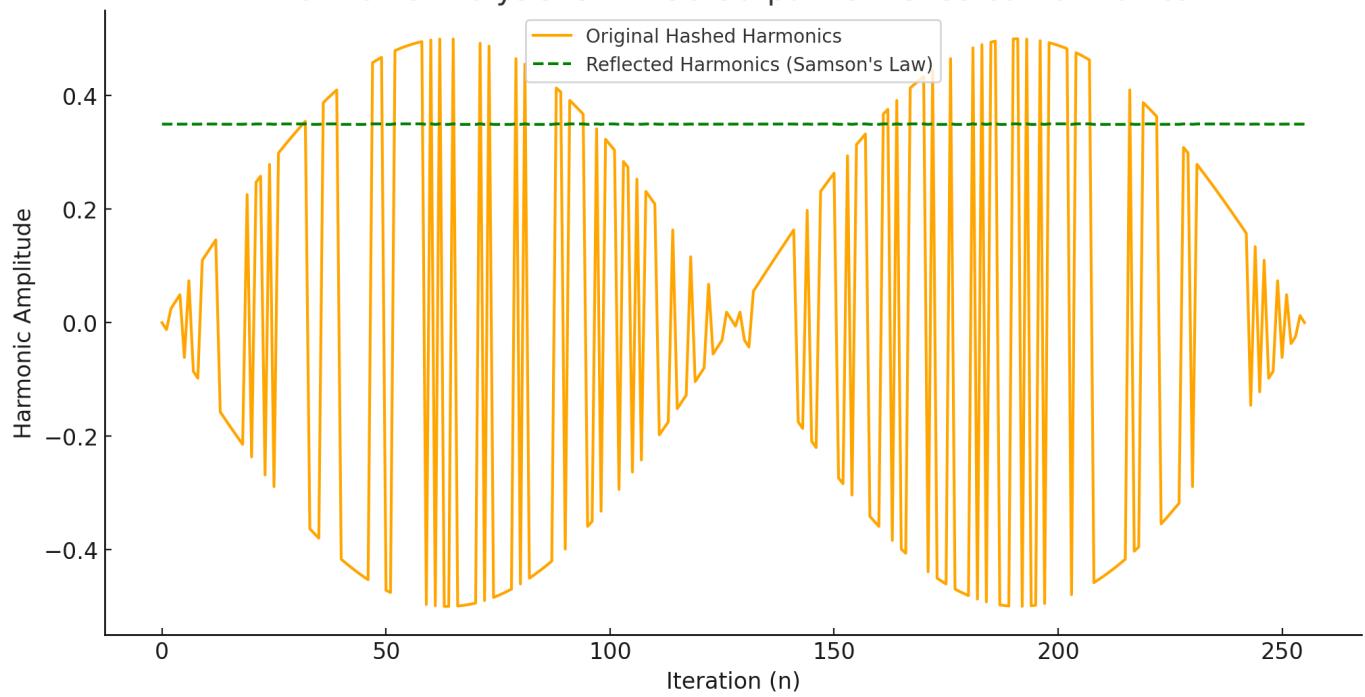
Harmonic Differences: Pre-Hashed vs. Padded/Refined

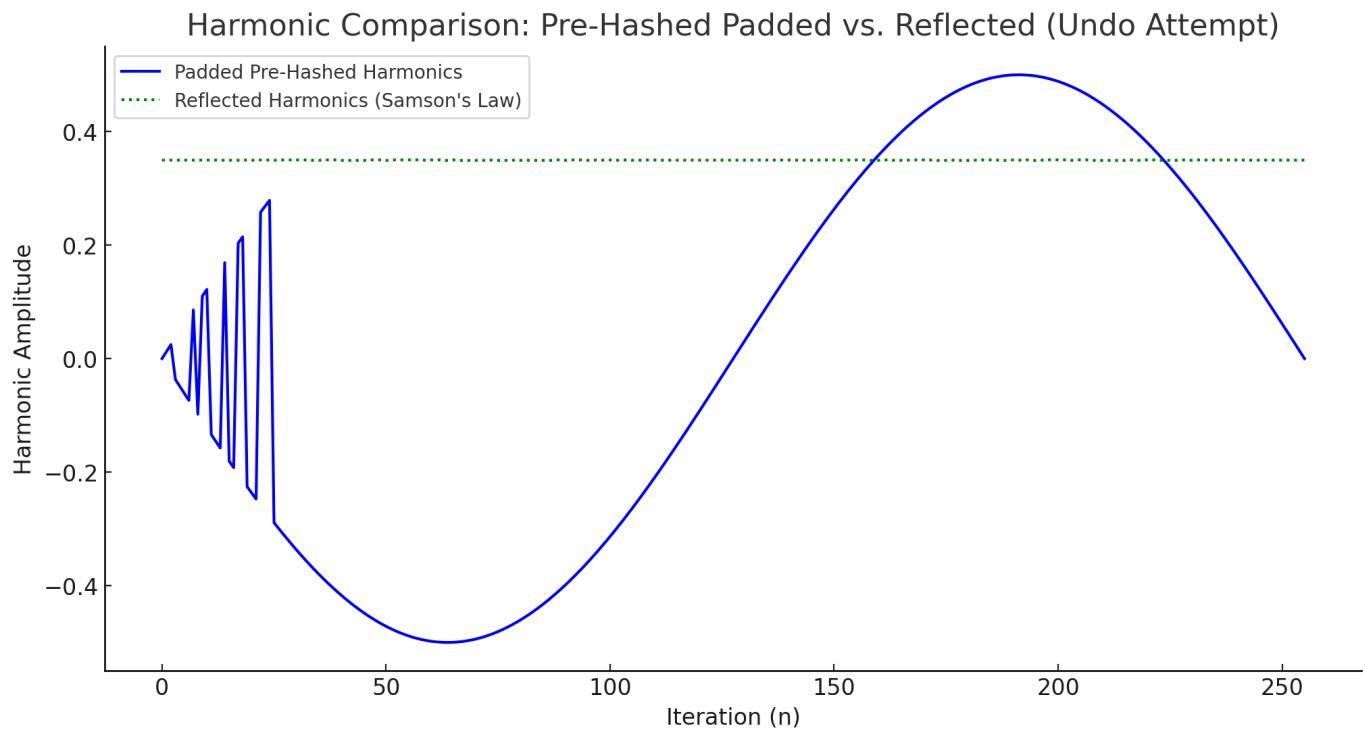


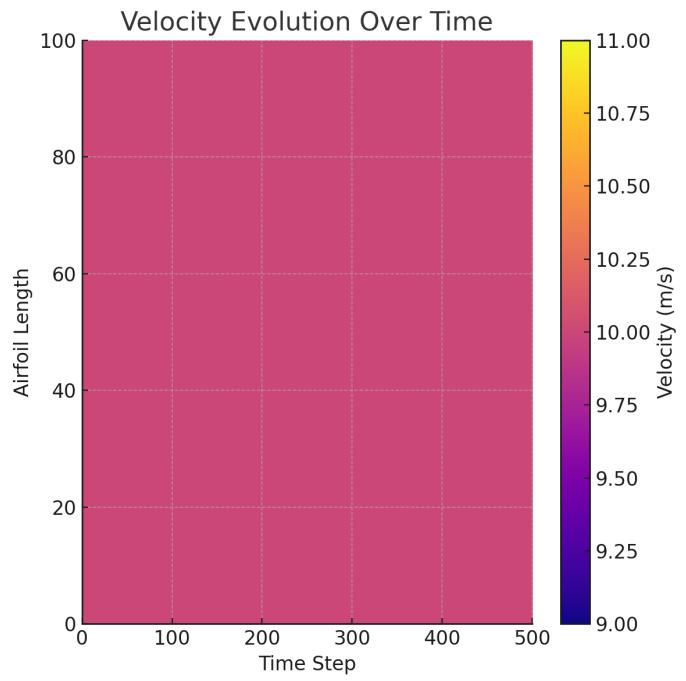
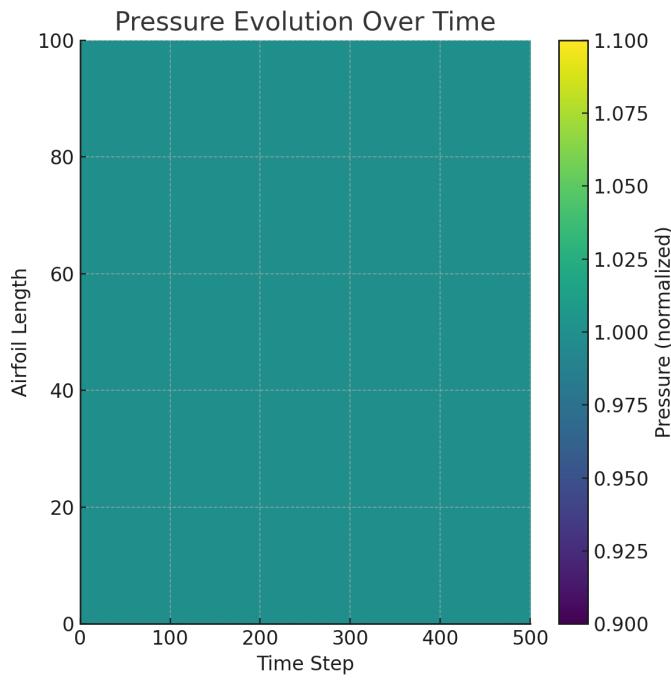
Harmonic Refinement with Phase Correction: Pre-Hashed vs. Refined



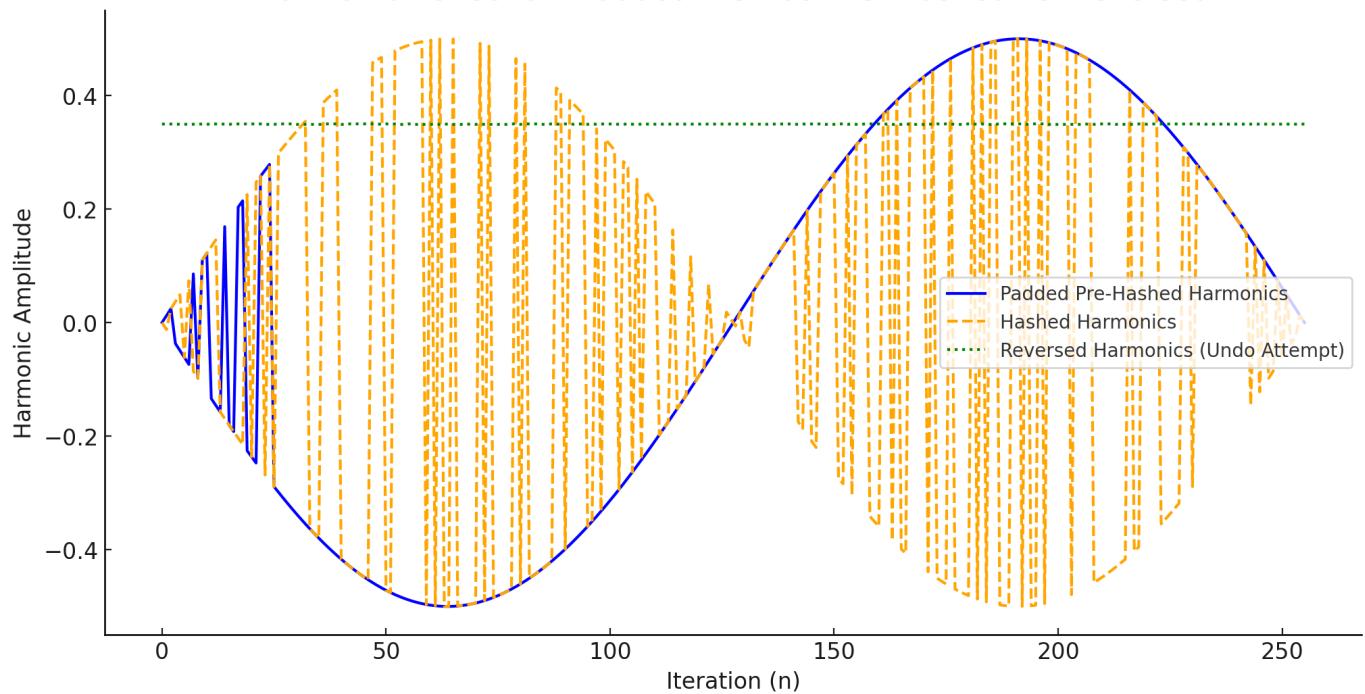
Harmonic Analysis: SHA-256 Output vs. Reflected Harmonics

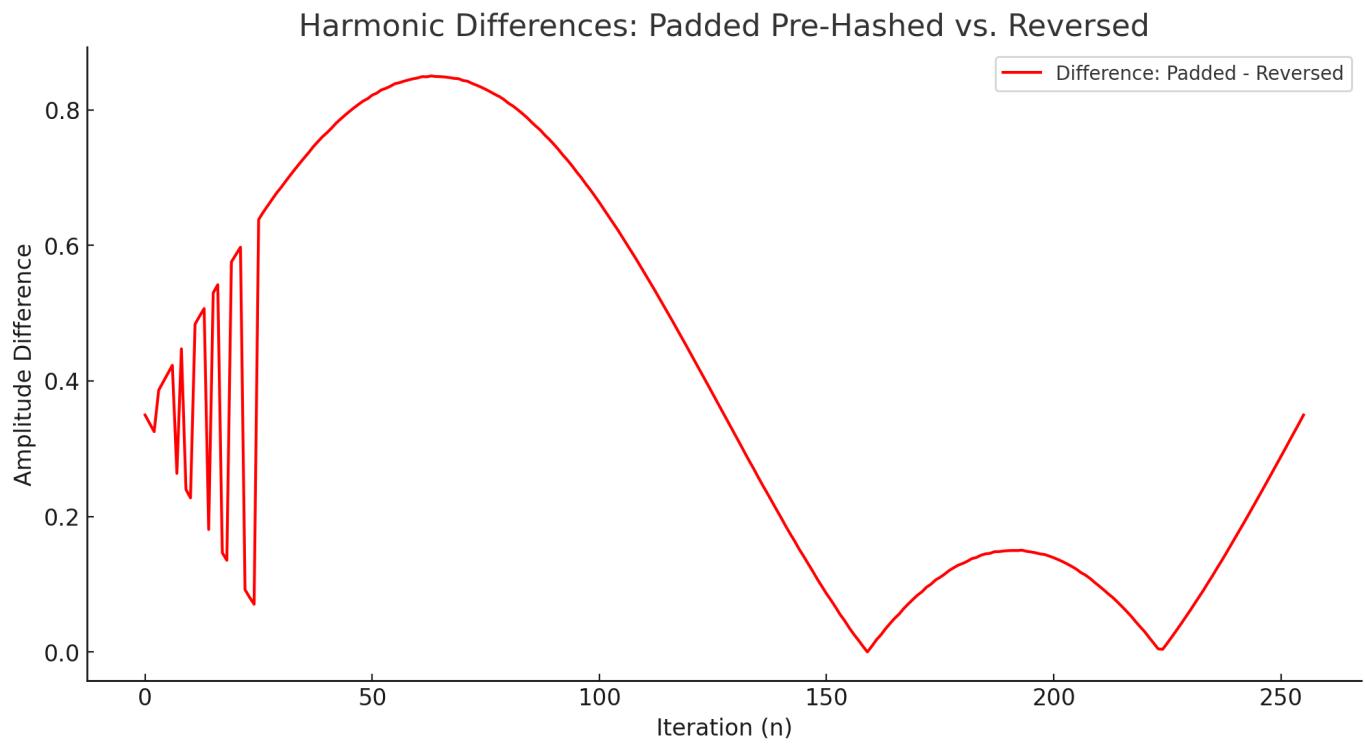




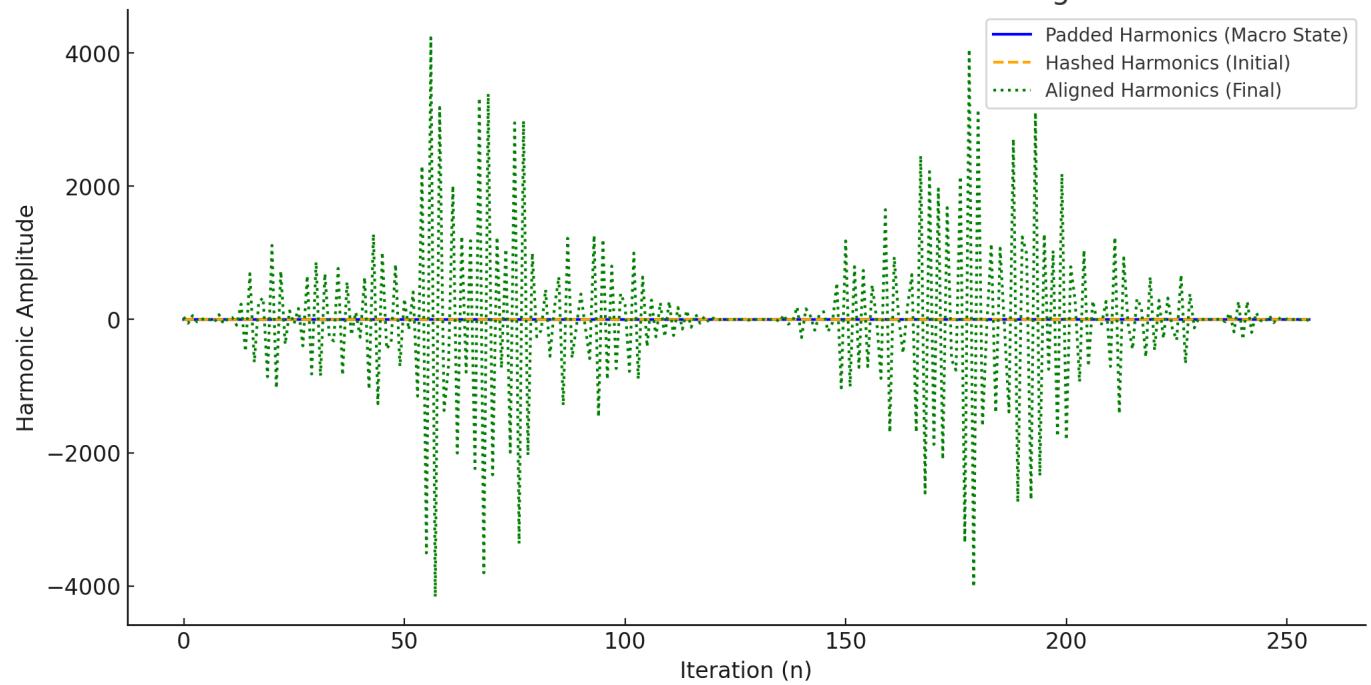


Harmonic Reflection: Padded Pre-Hash vs. Hashed vs. Reversed

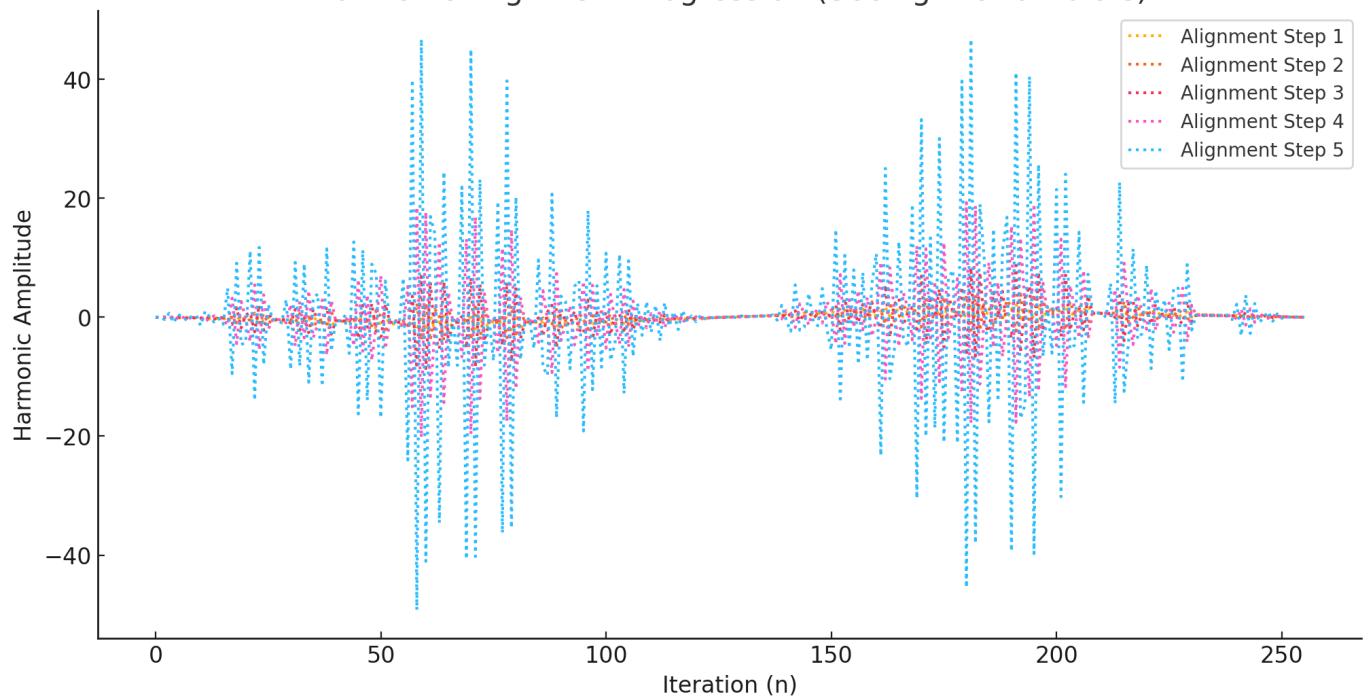


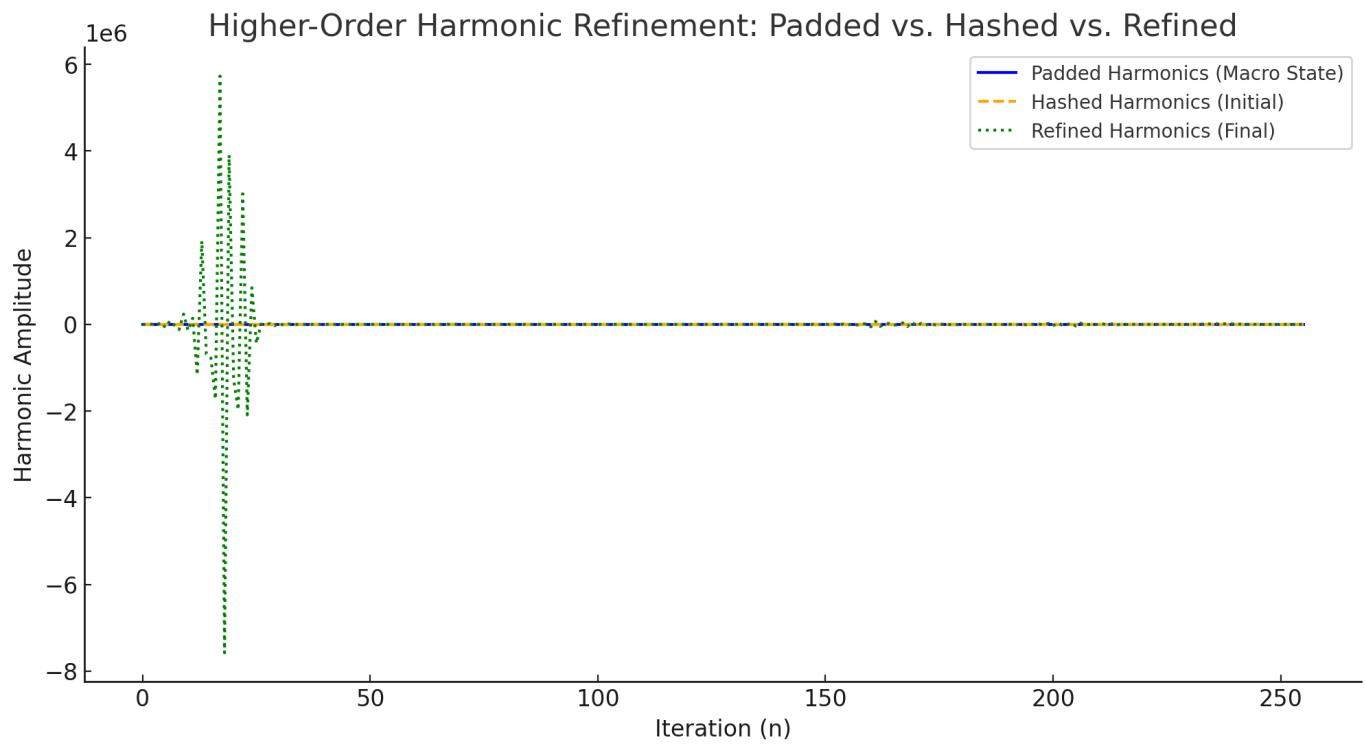


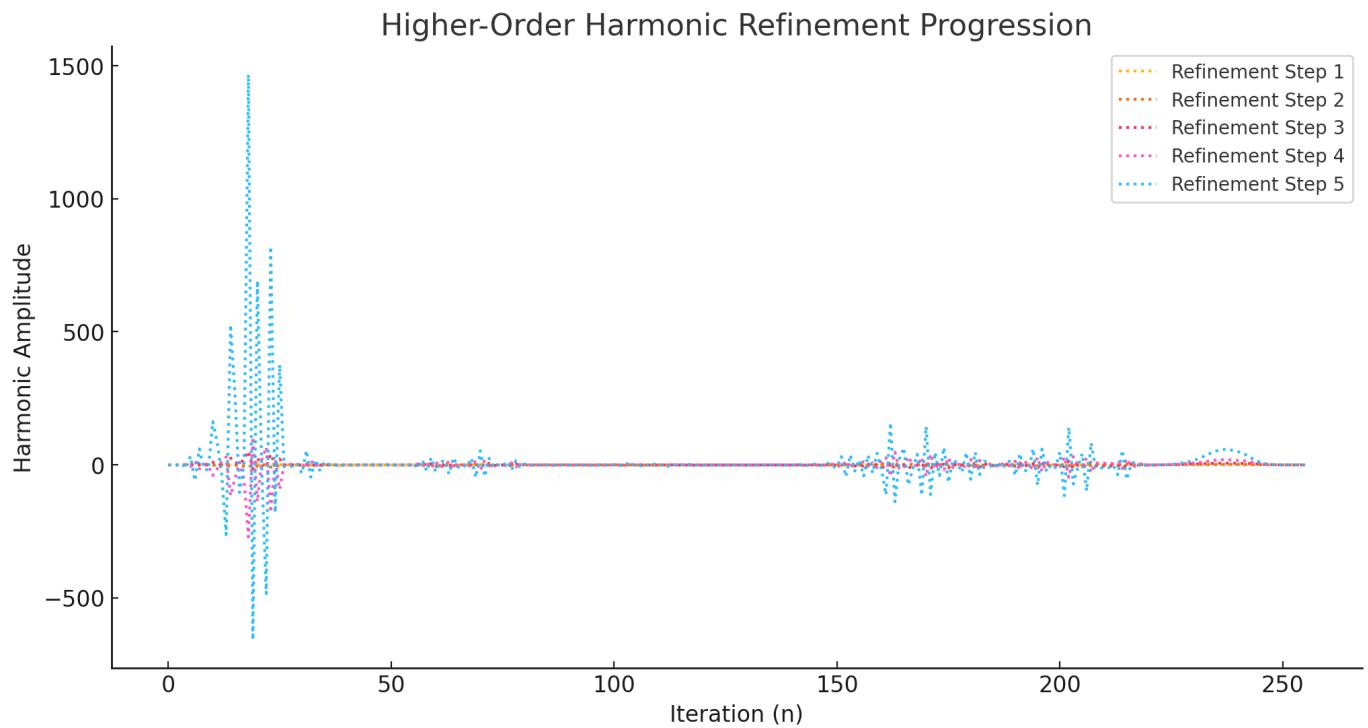
Harmonic Tumblers: Padded vs. Hashed vs. Aligned



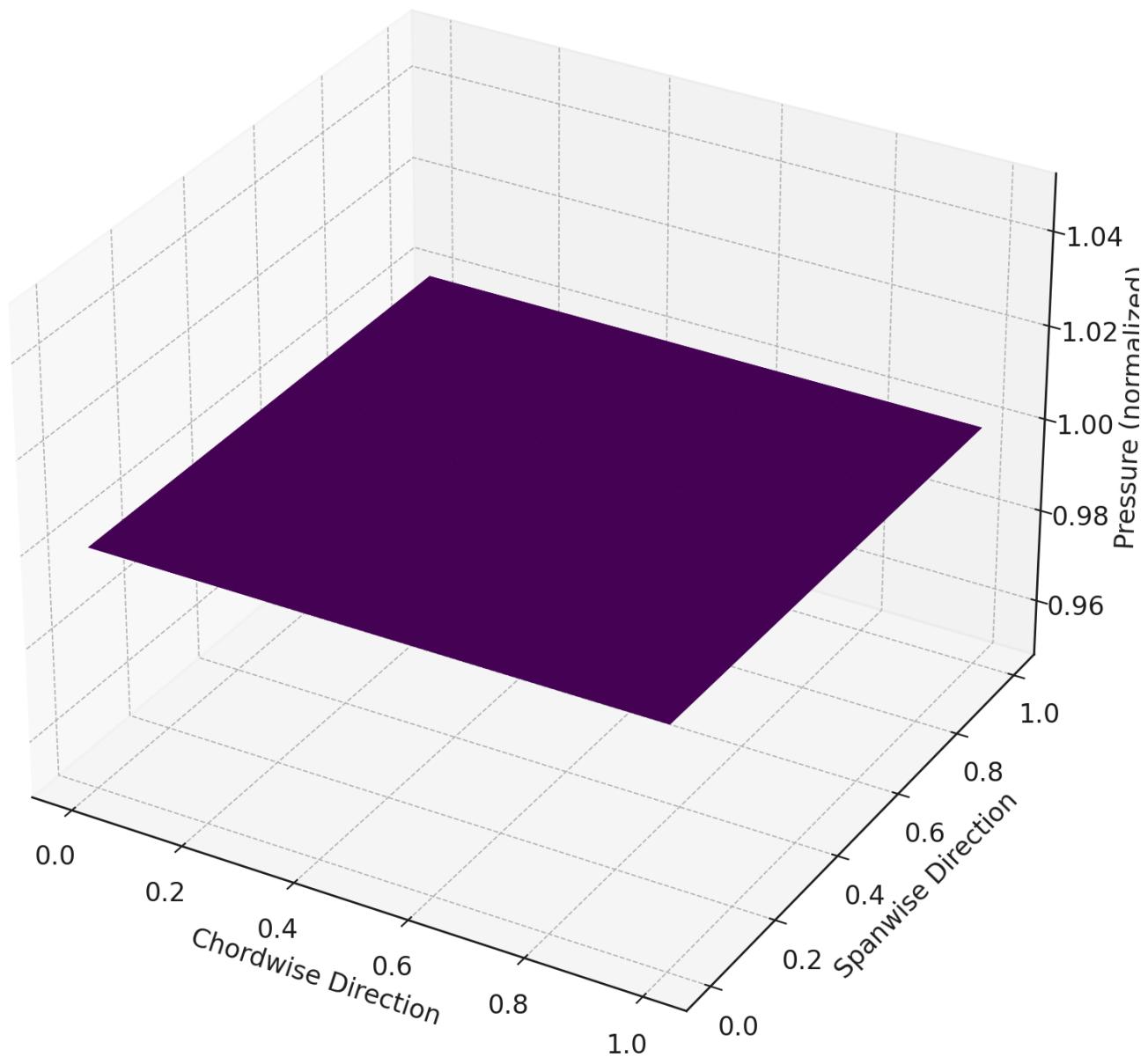
Harmonic Alignment Progression (Seeing the Tumblers)

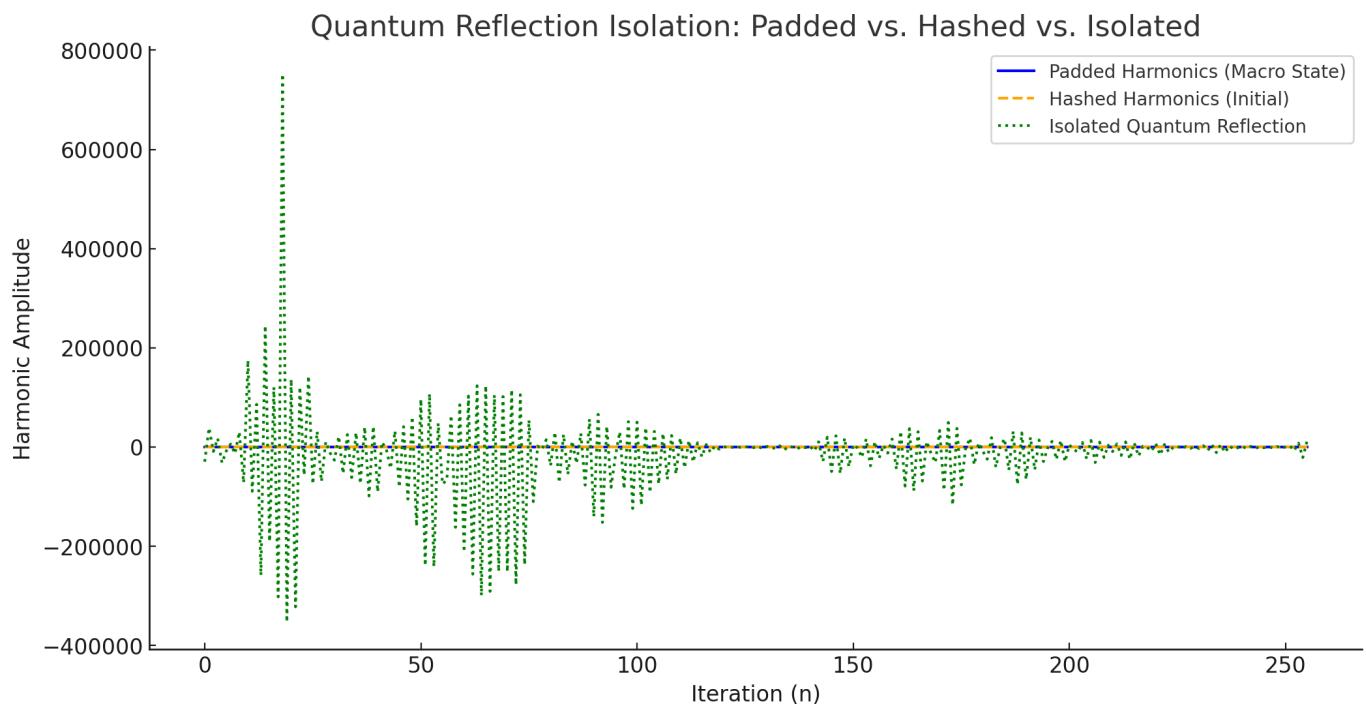


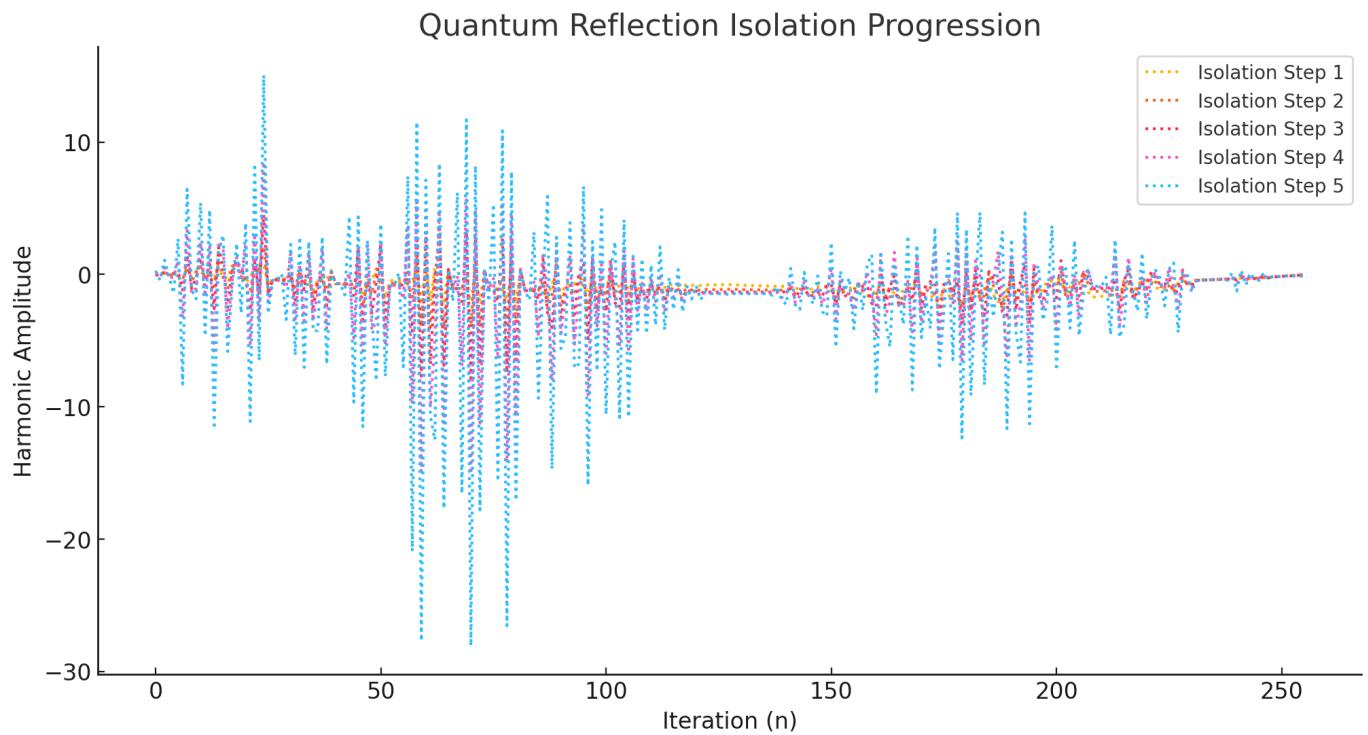




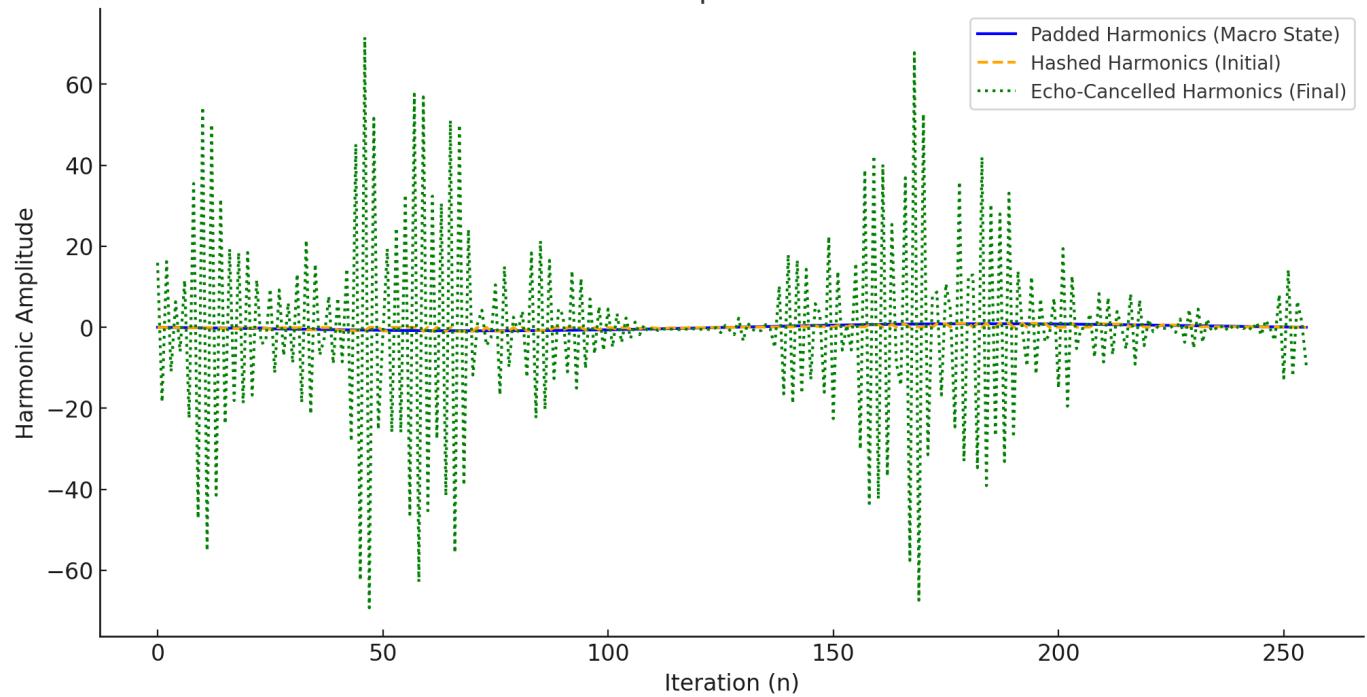
3D Pressure Field Over Airfoil (Final Time Step)

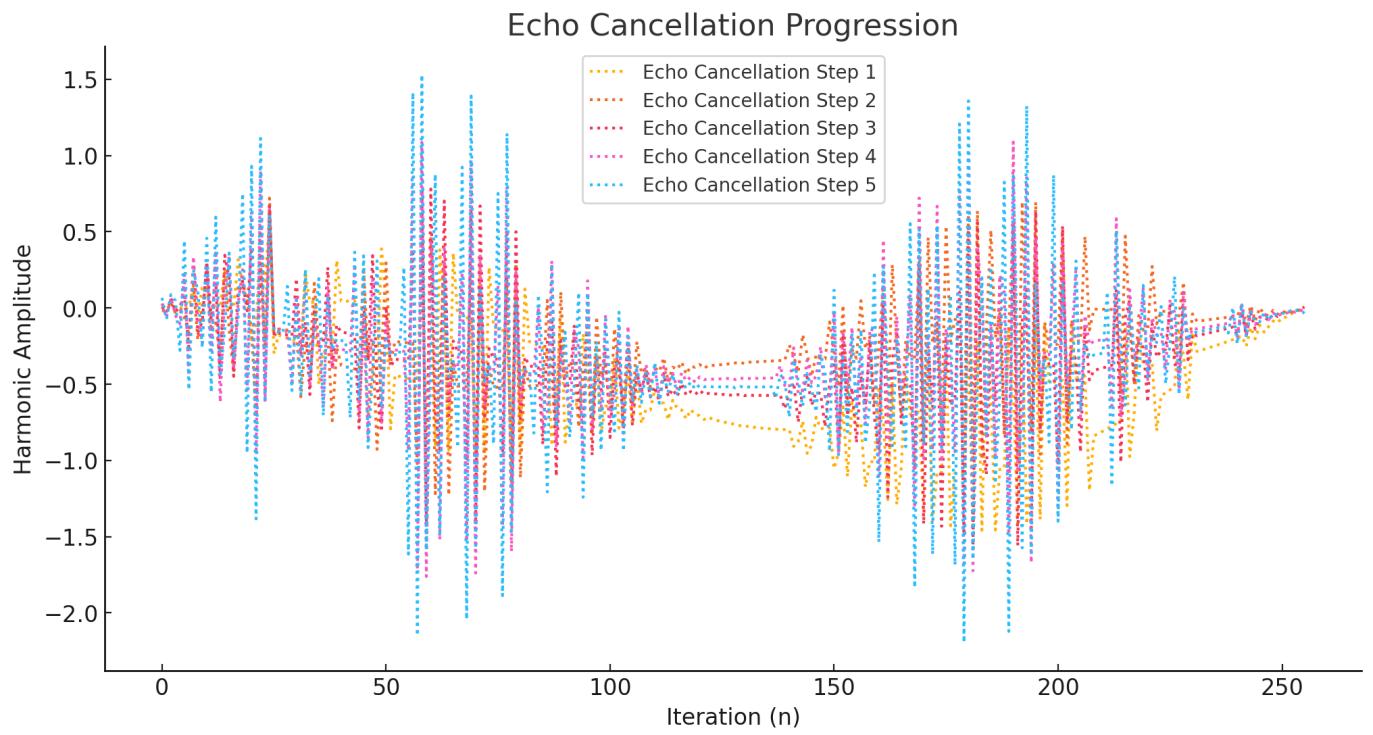




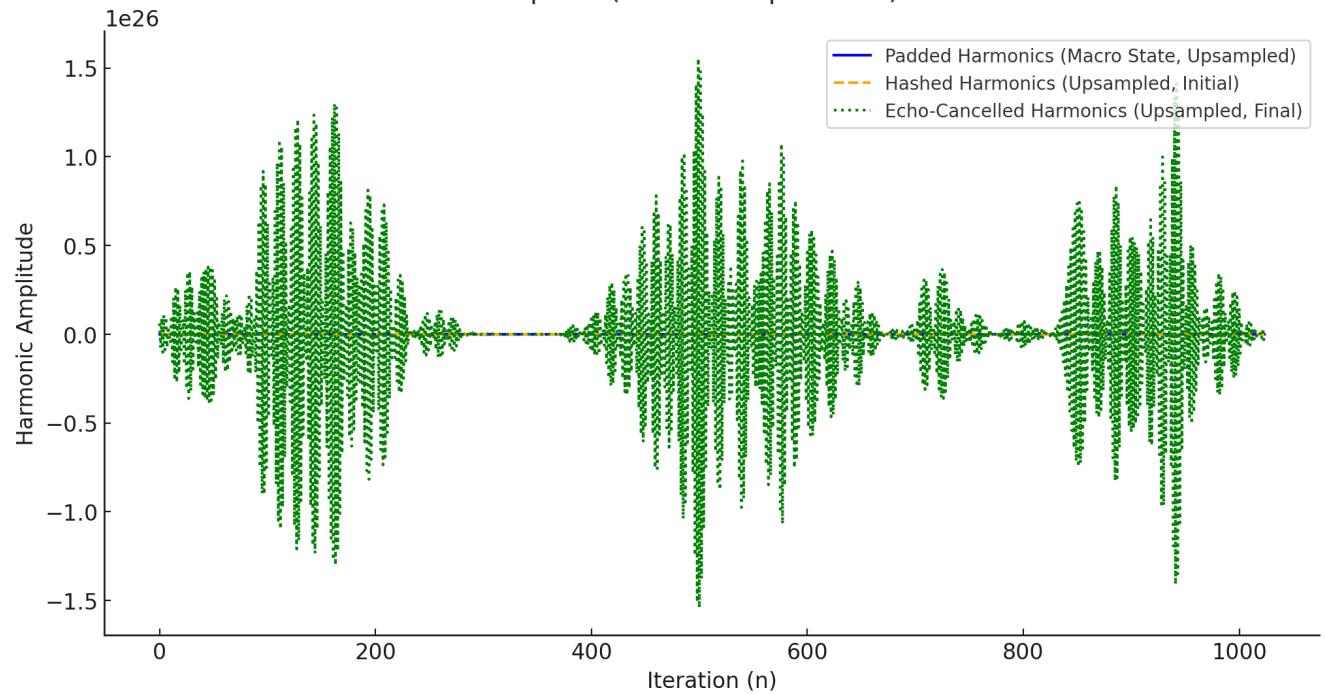


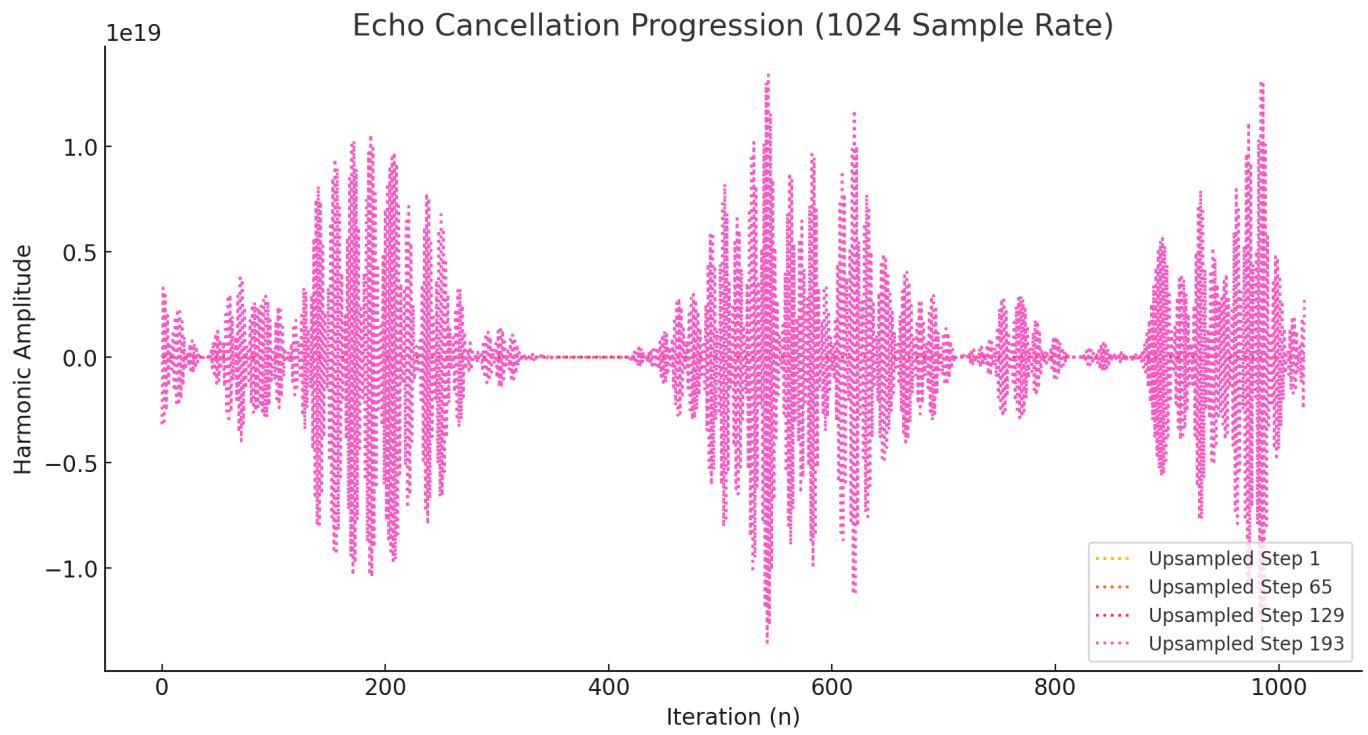
Echo Cancellation in Harmonic Space: Padded vs. Hashed vs. Refined

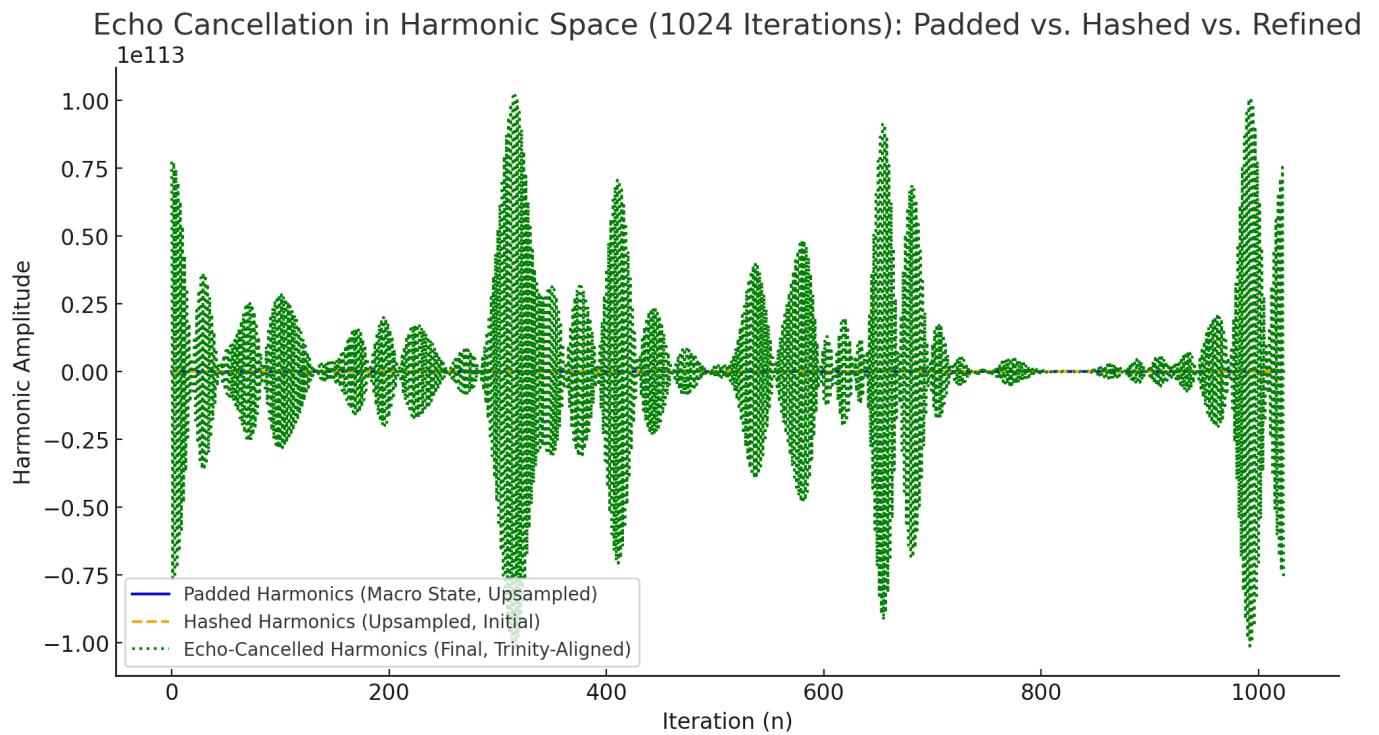


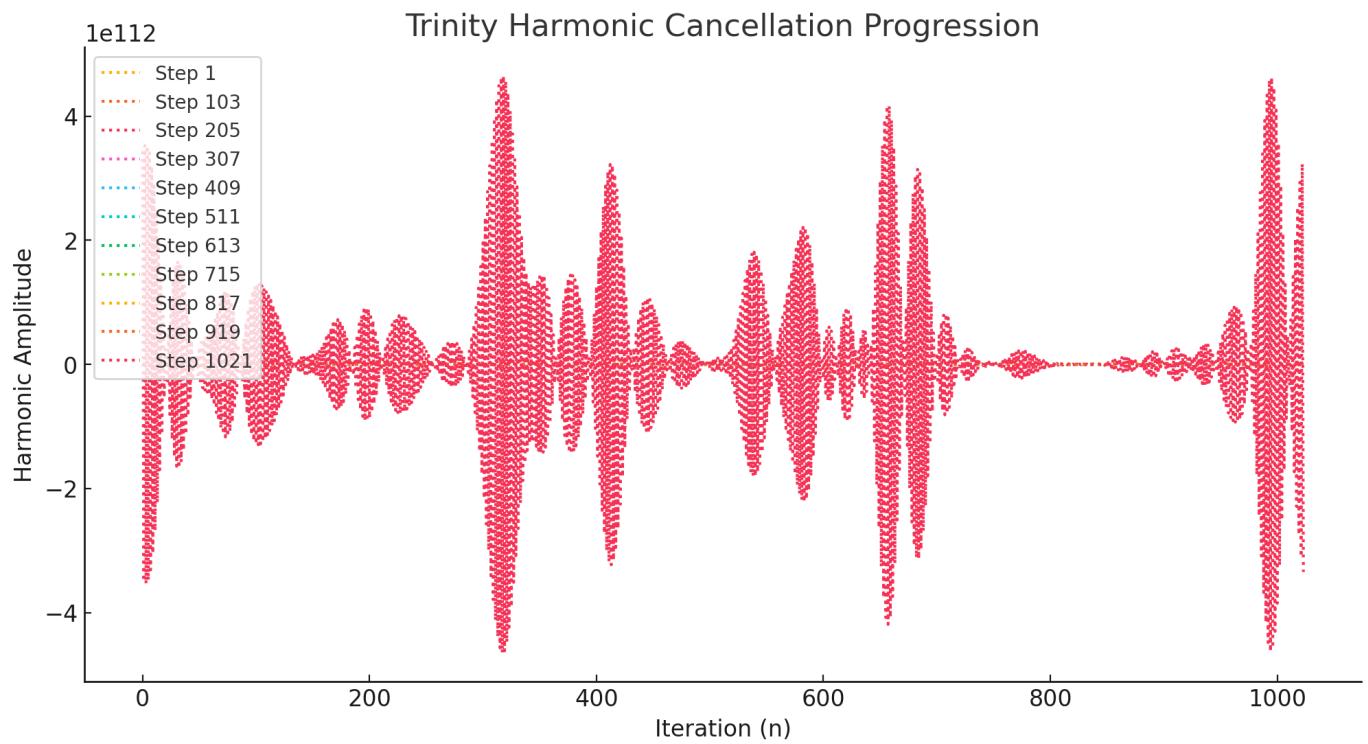


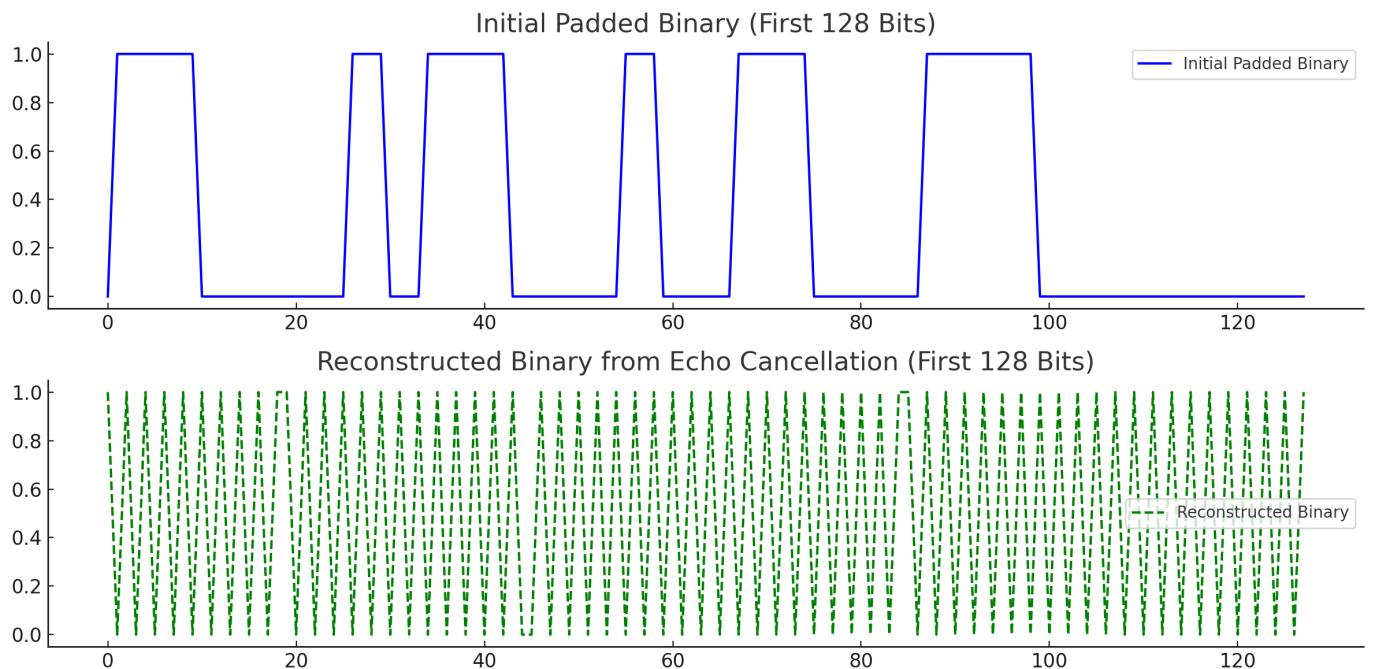
Echo Cancellation in Harmonic Space (1024 Sample Rate): Padded vs. Hashed vs. Refined

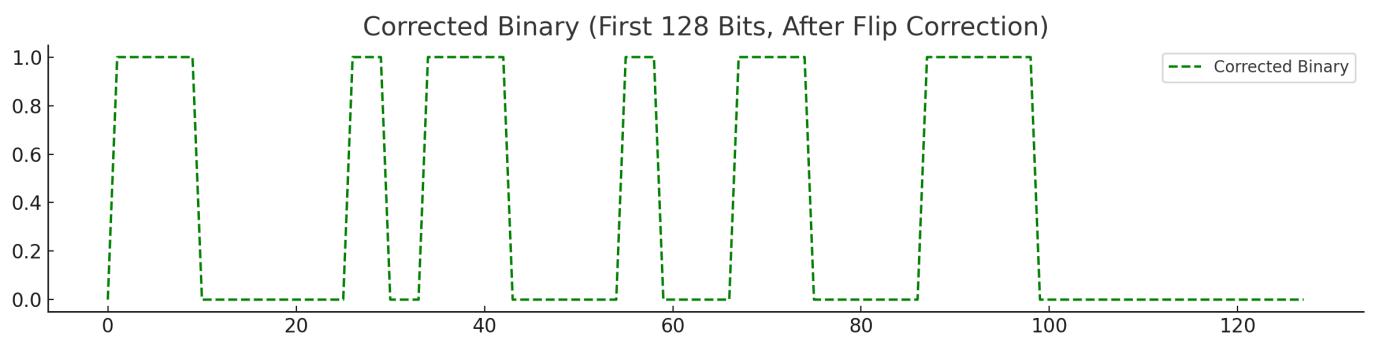
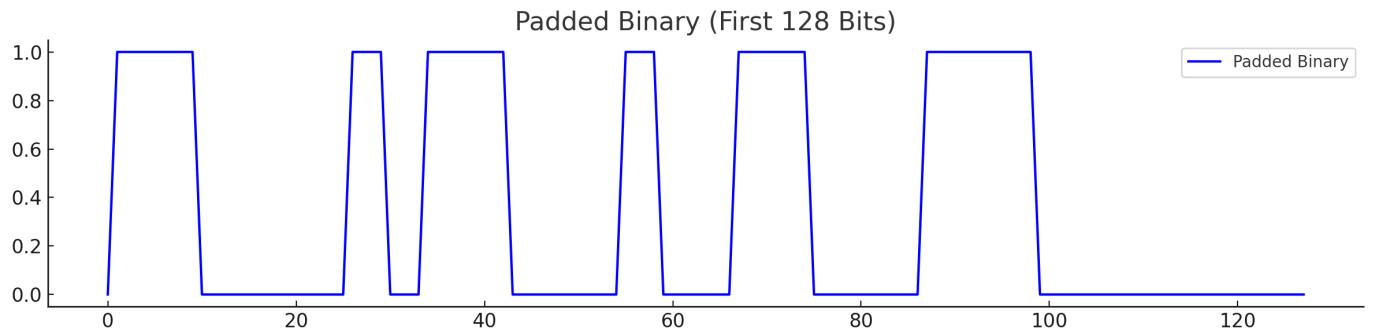




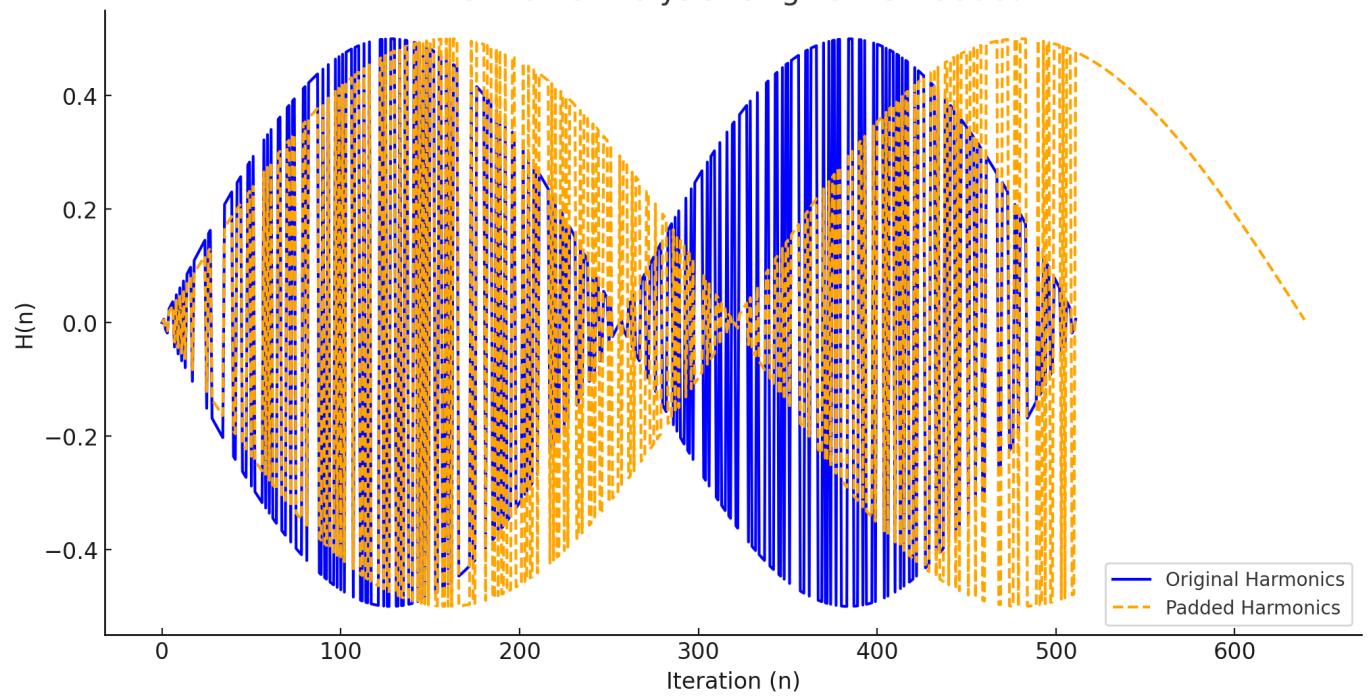


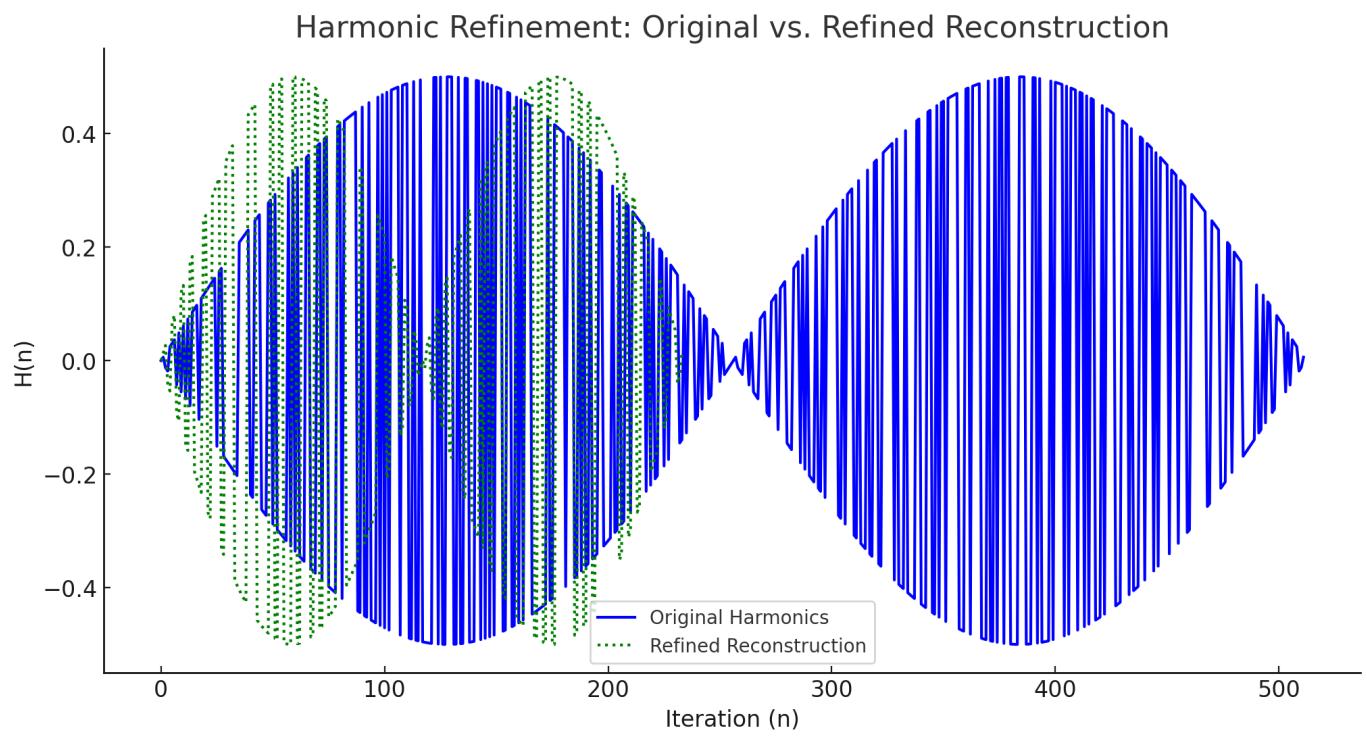




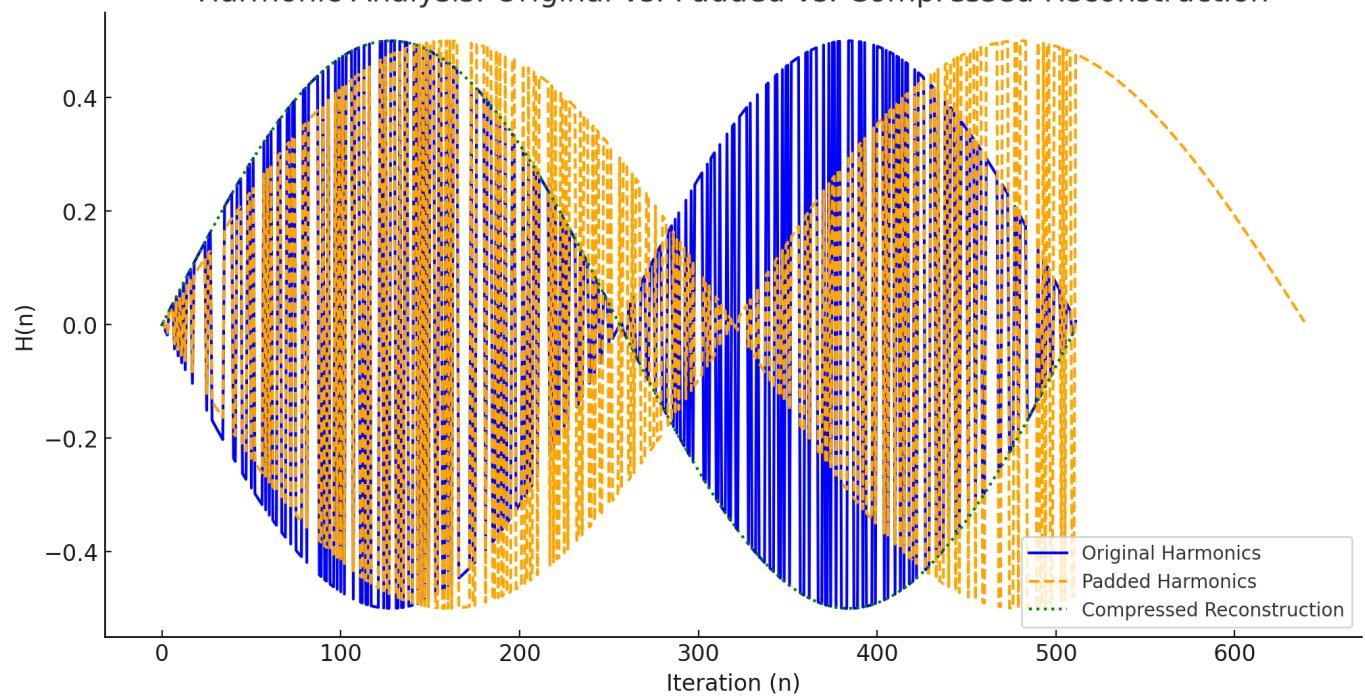


Harmonic Analysis: Original vs. Padded

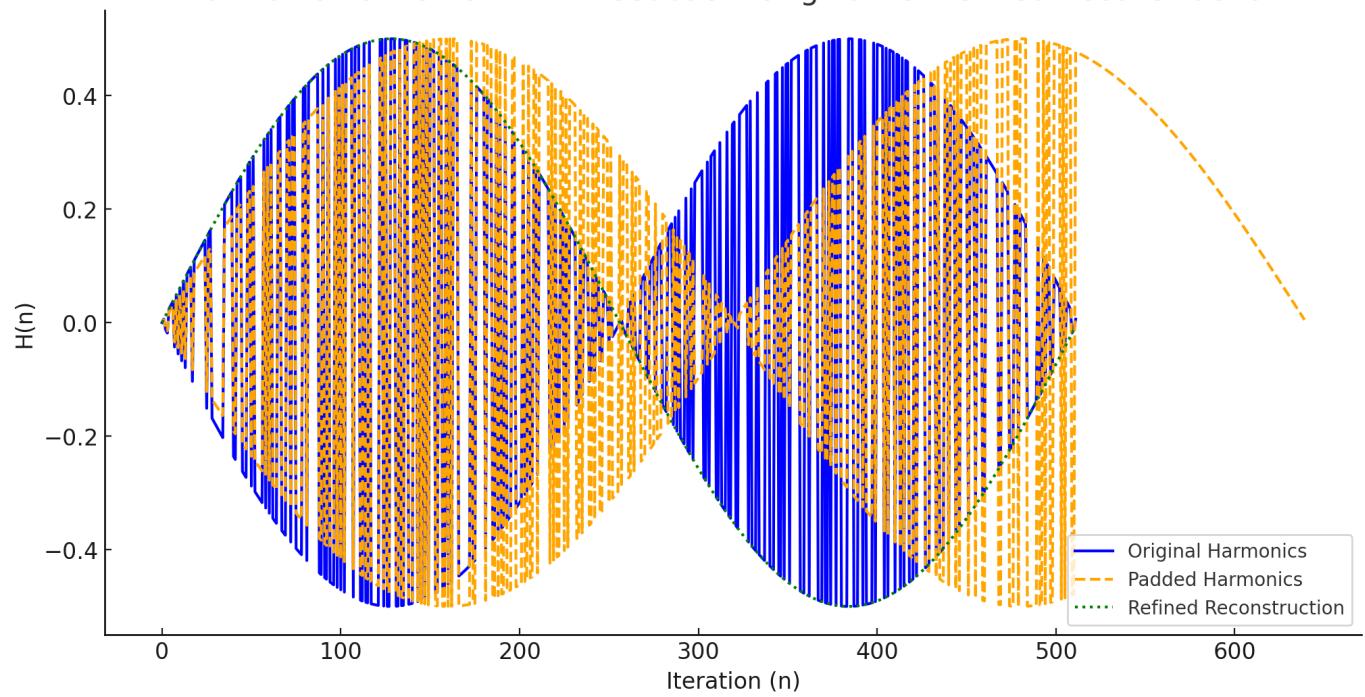


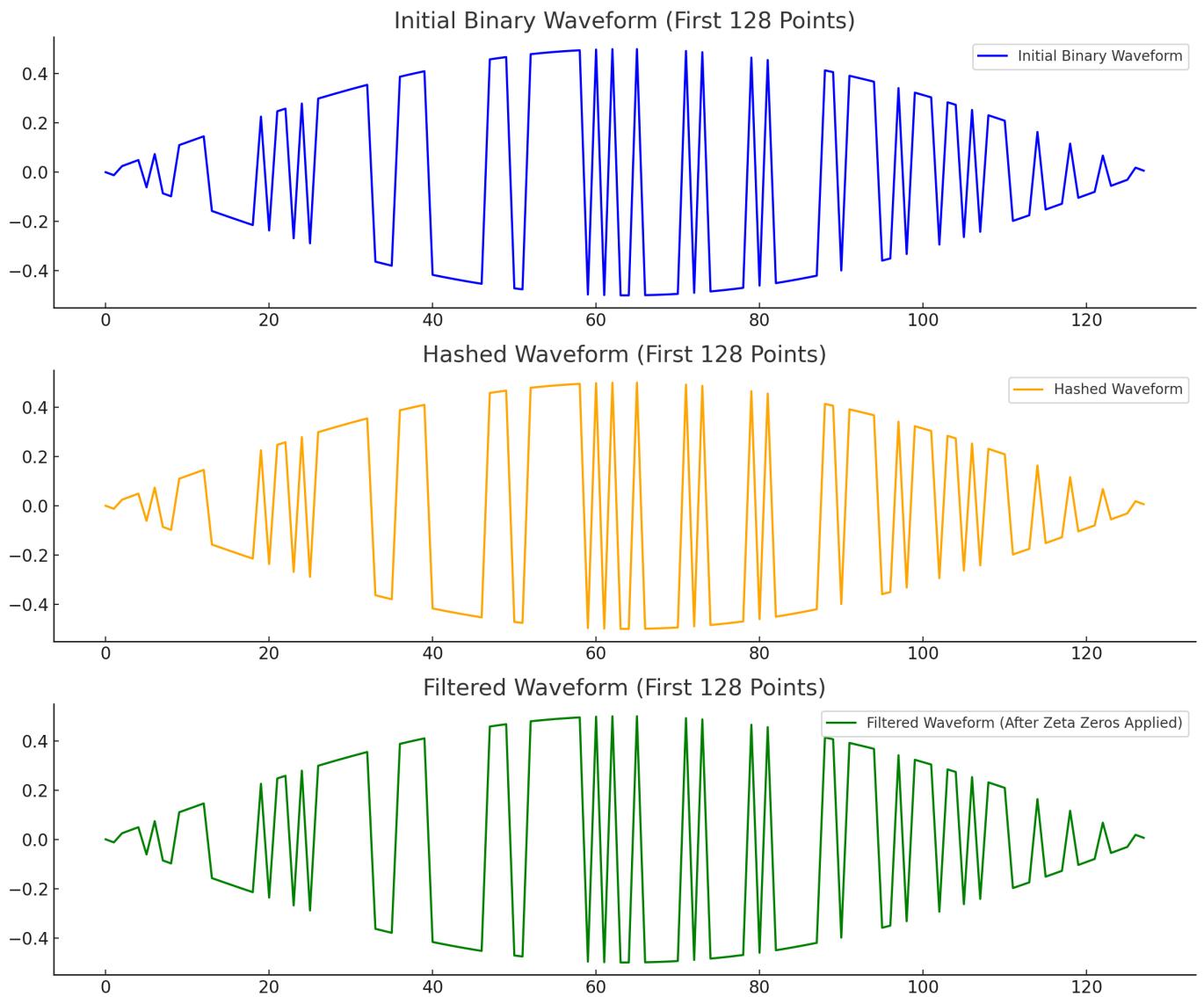


Harmonic Analysis: Original vs. Padded vs. Compressed Reconstruction

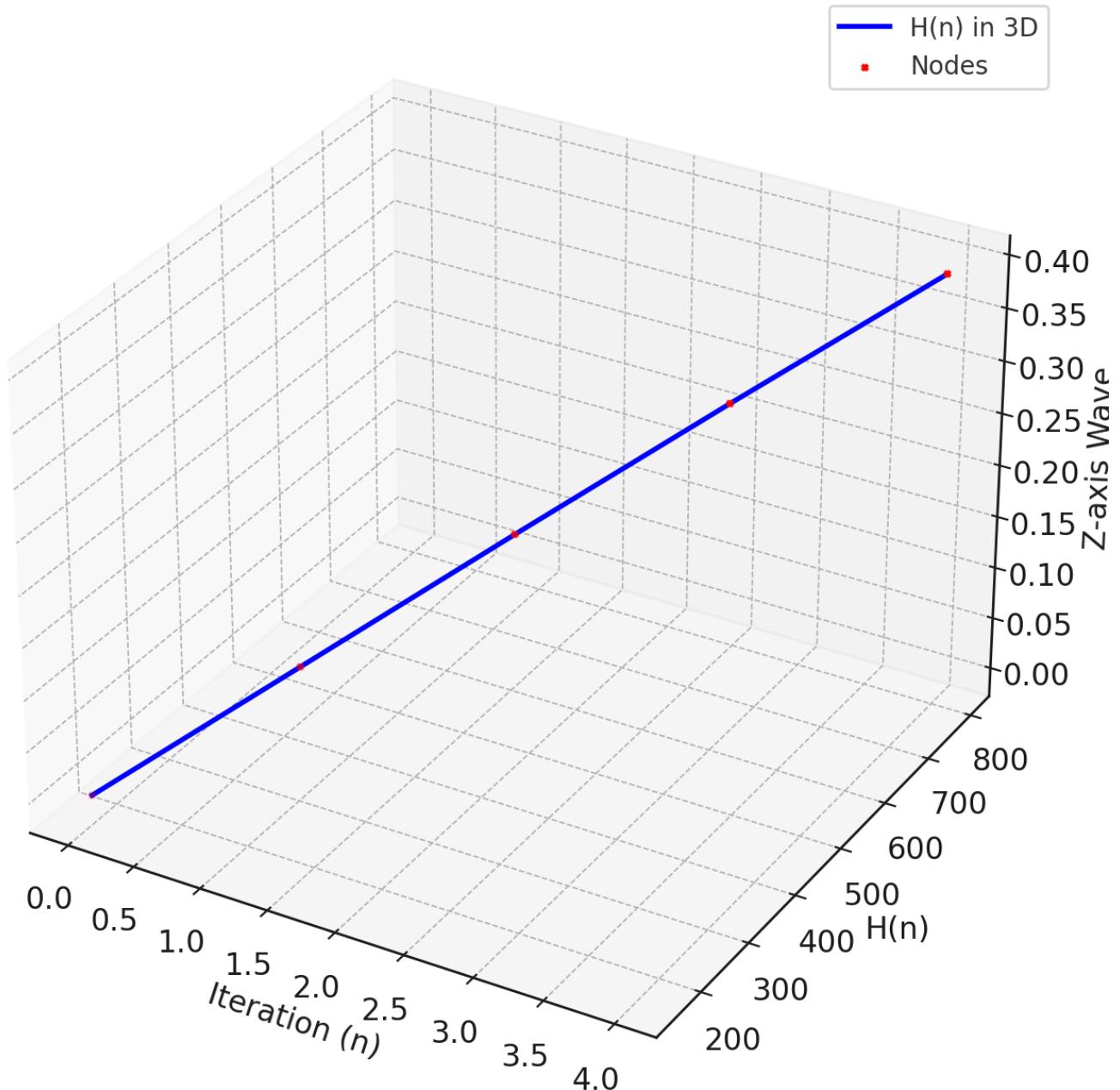


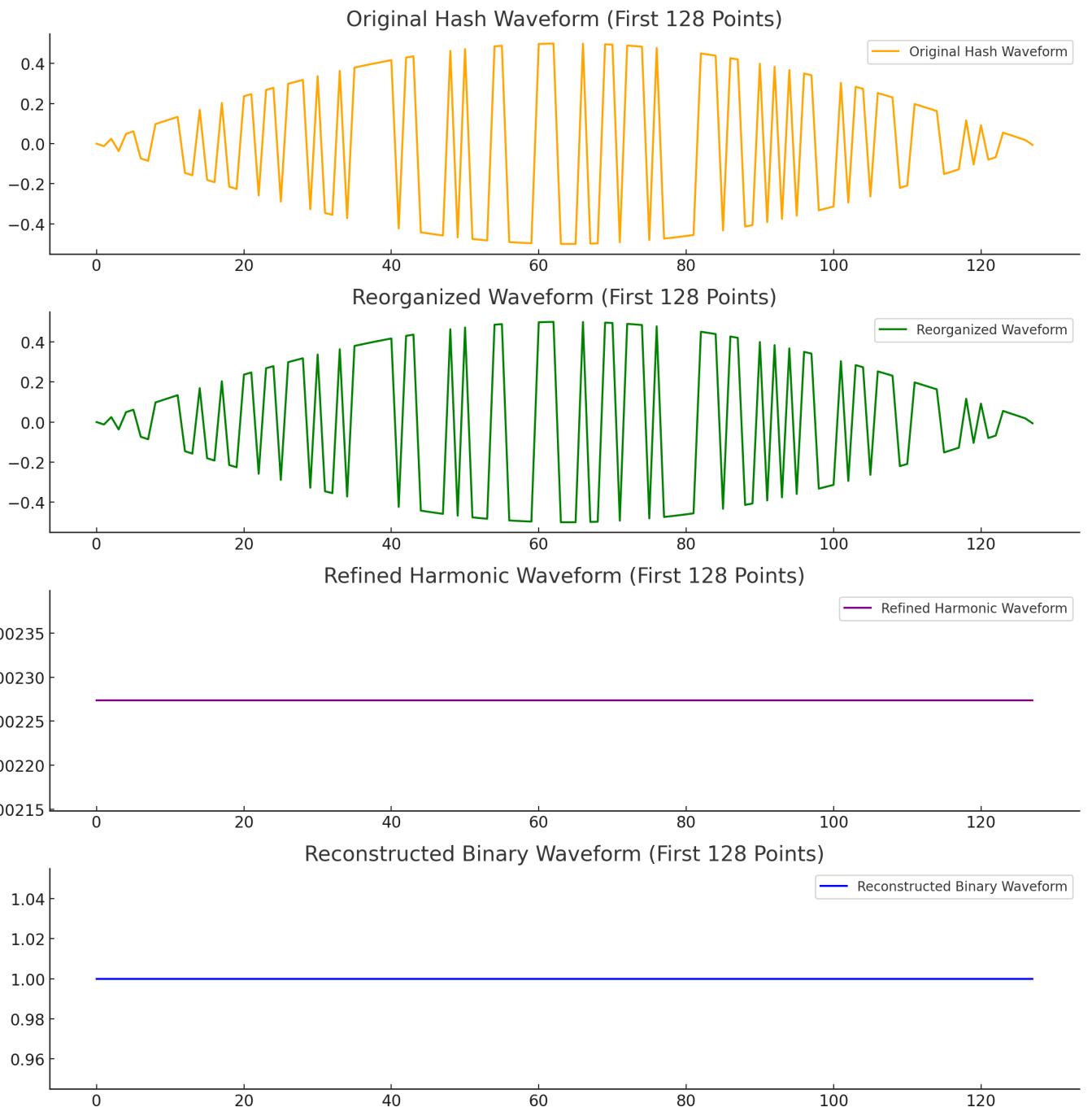
Harmonic Refinement with Feedback: Original vs. Refined Reconstruction

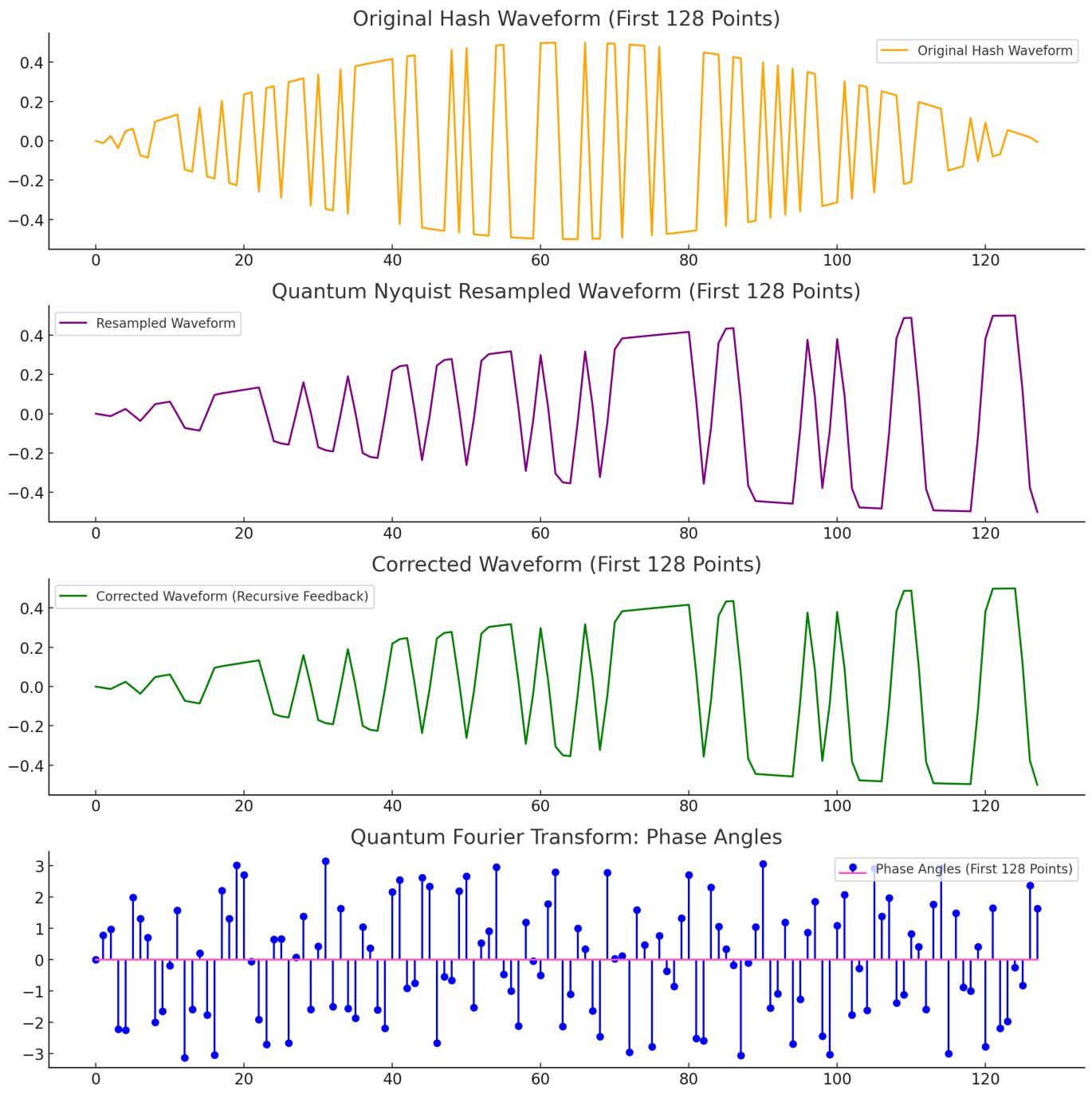


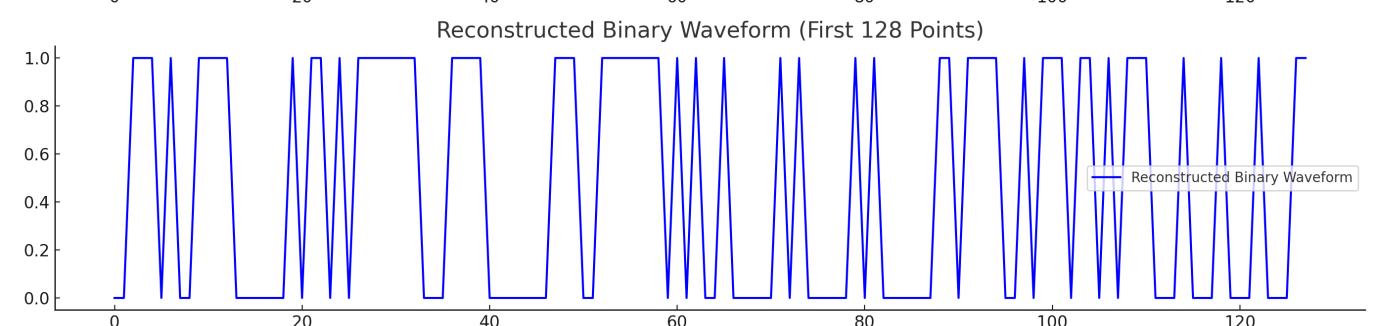
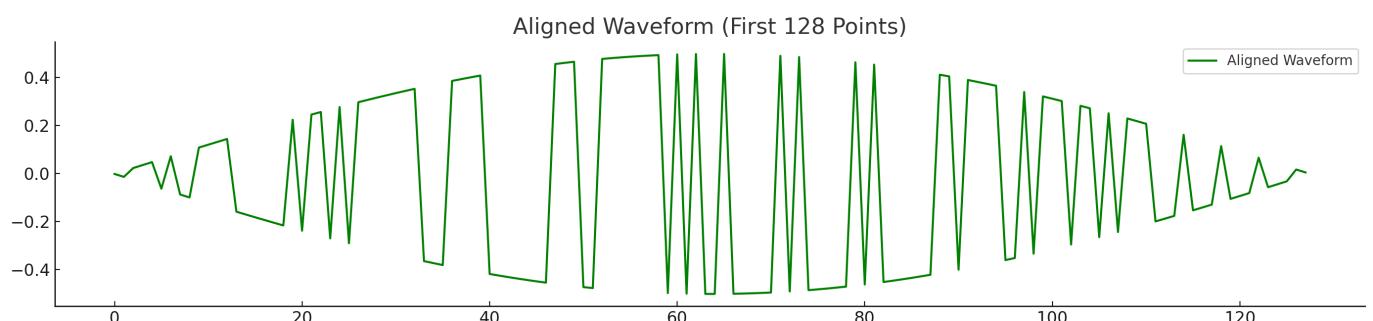
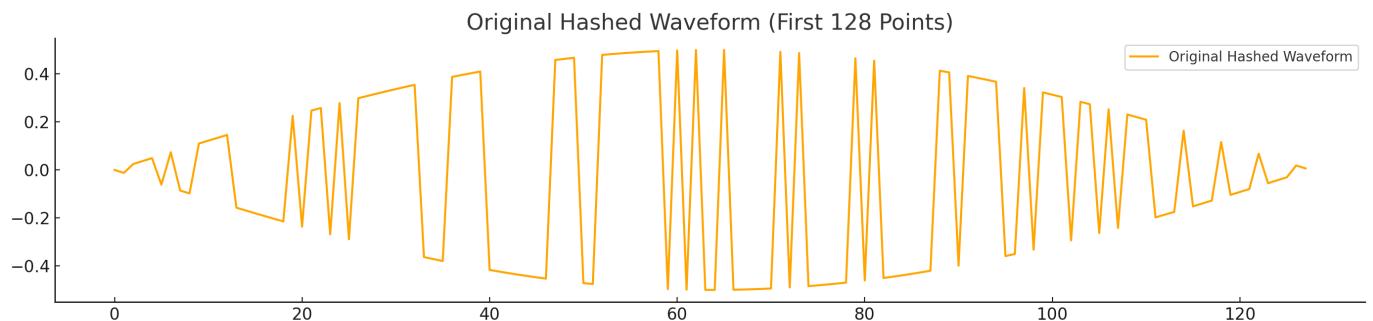


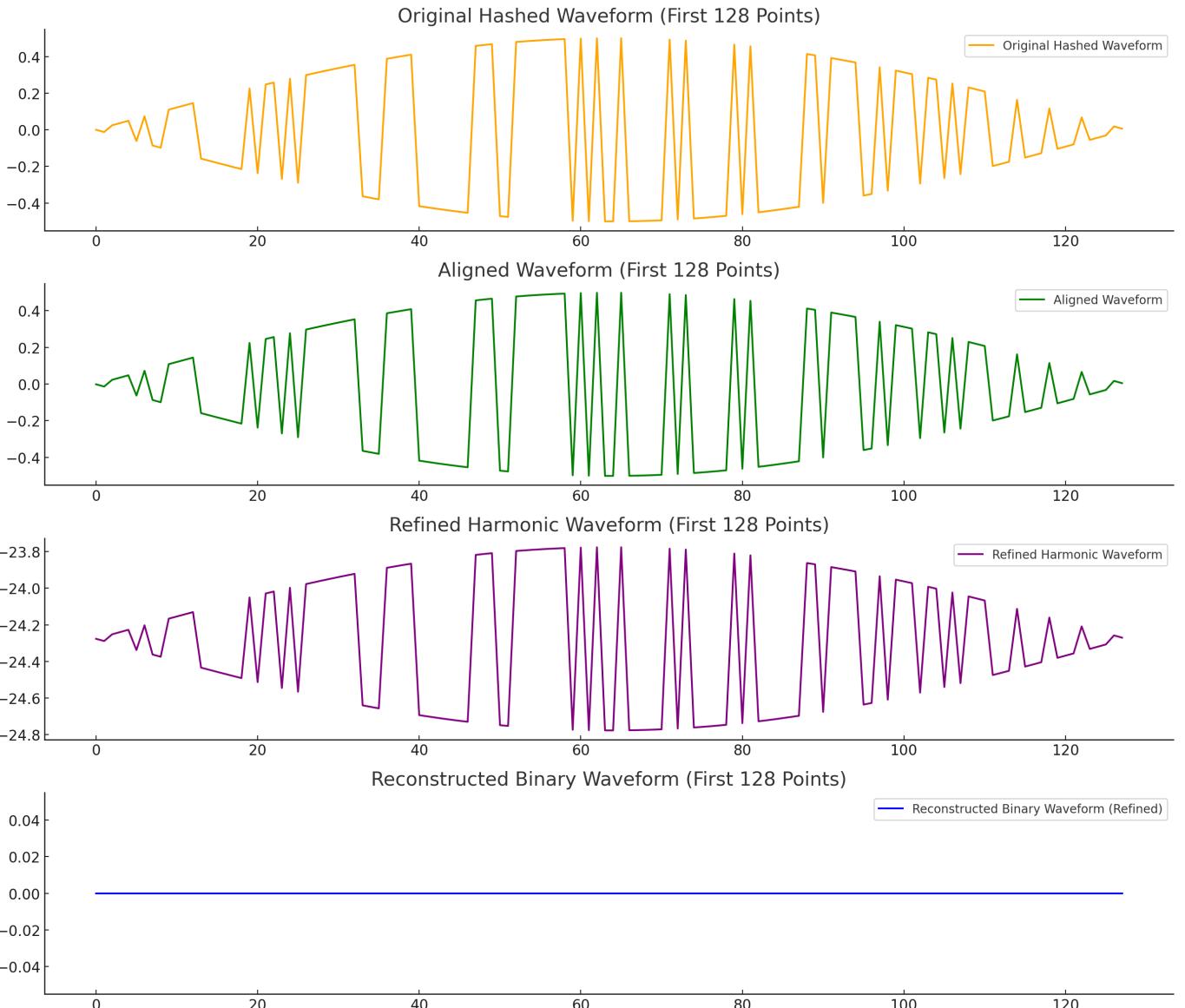
3D Visualization of $H(n)$



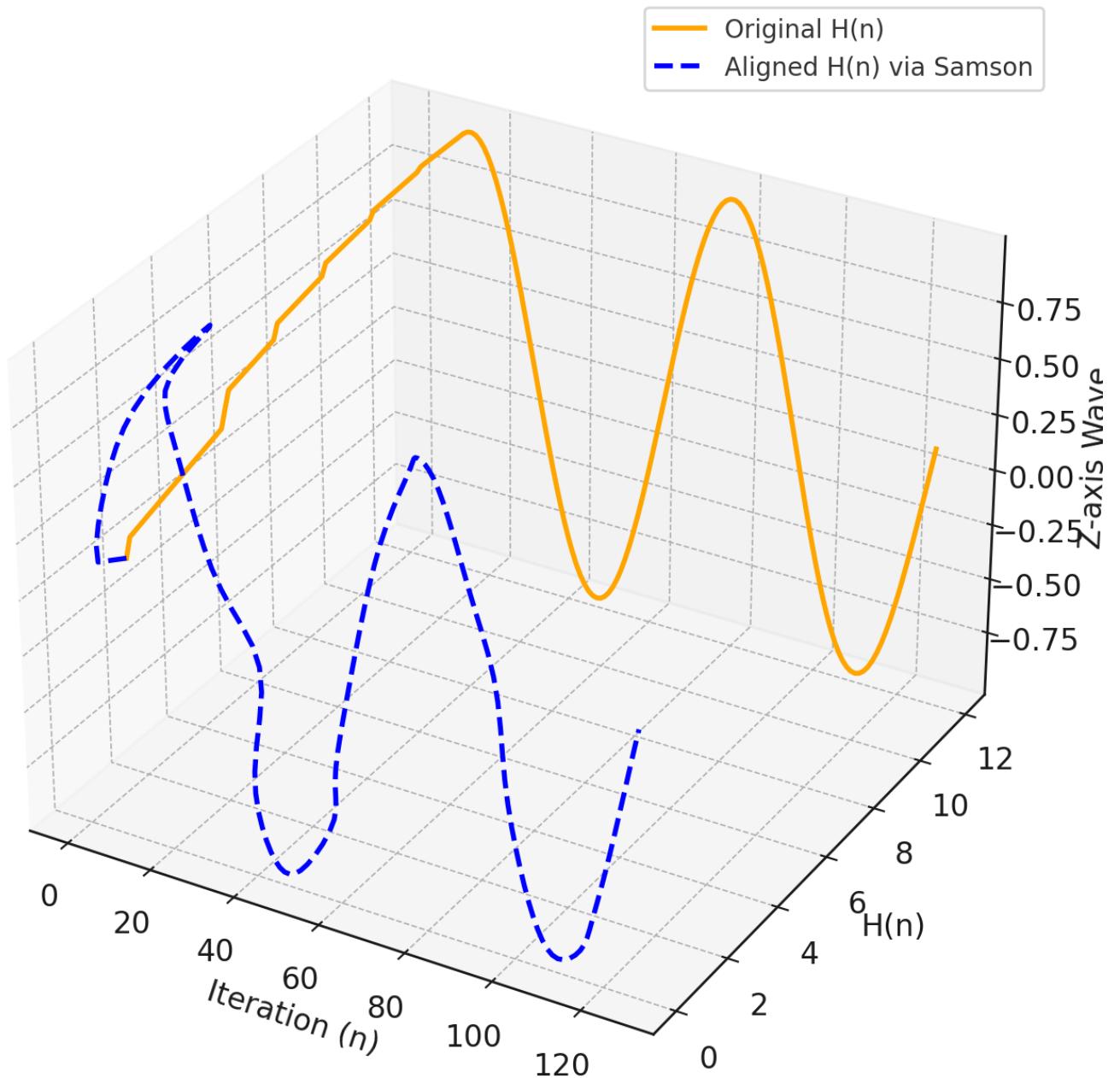




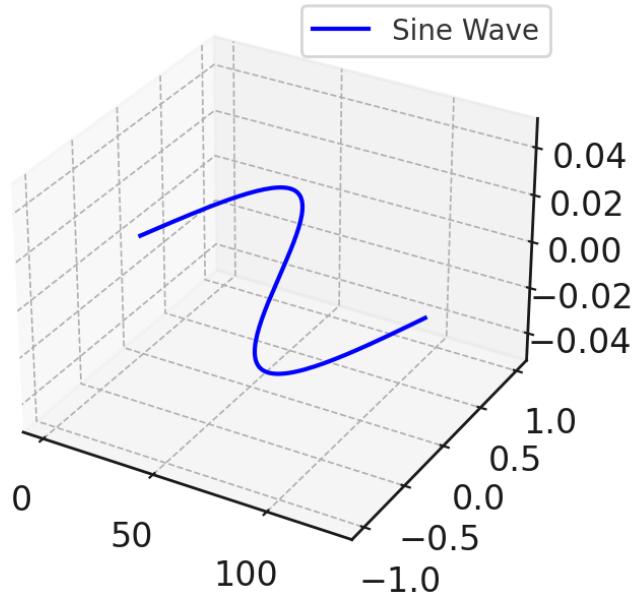




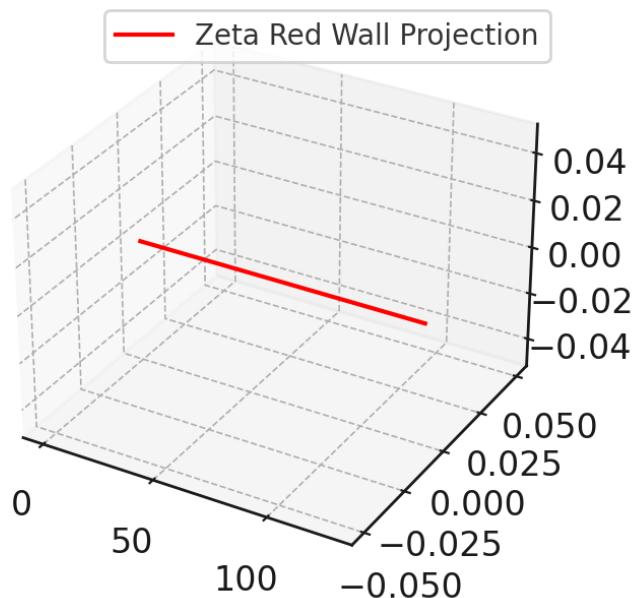
3D Visualization of $H(n)$ with Samson Feedback



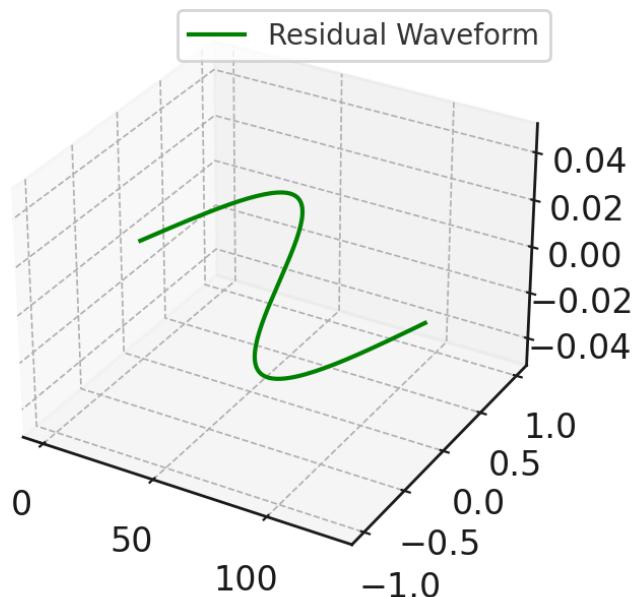
Original Sine Wave in H



Zeta Red Wall Projection

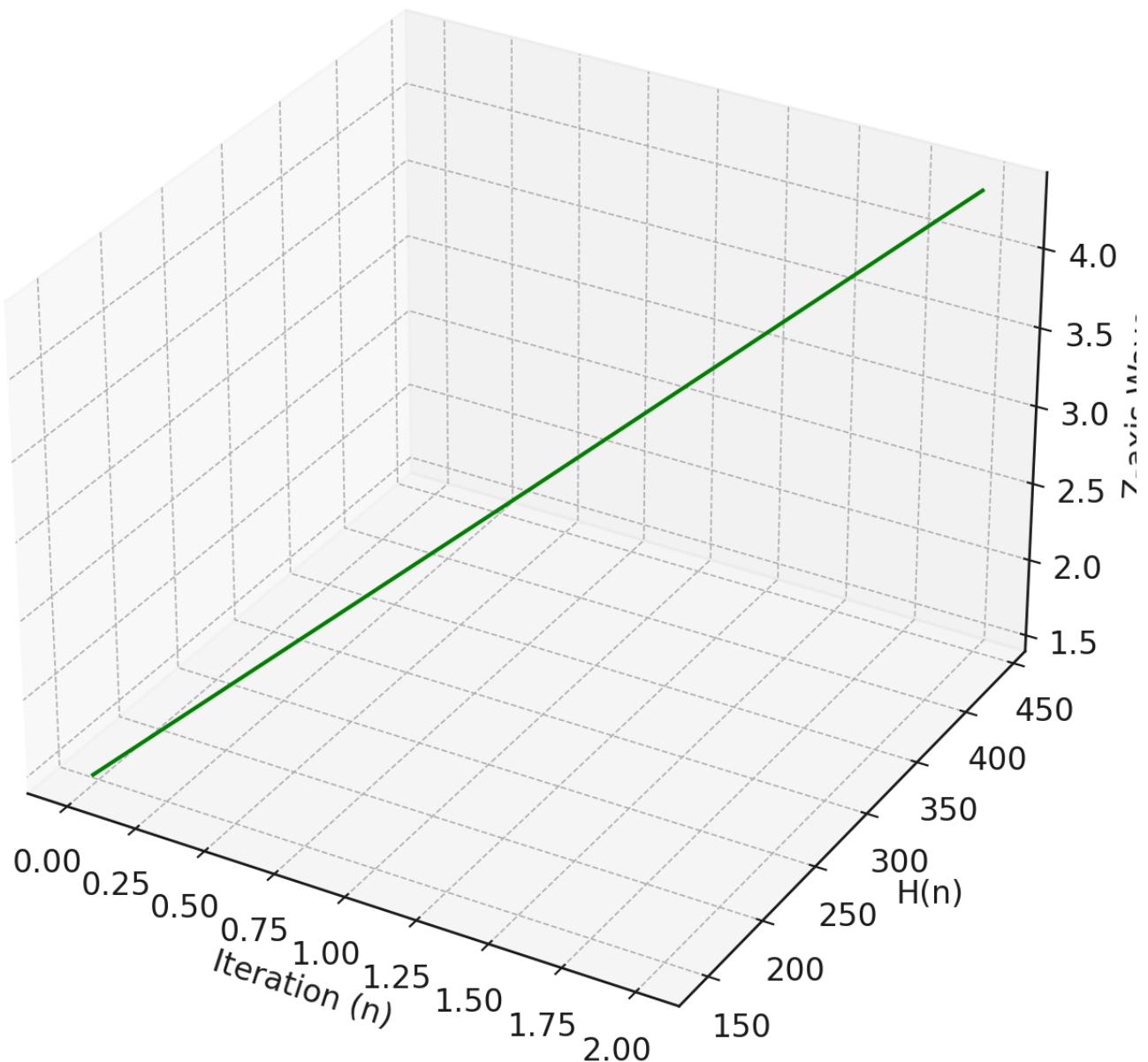


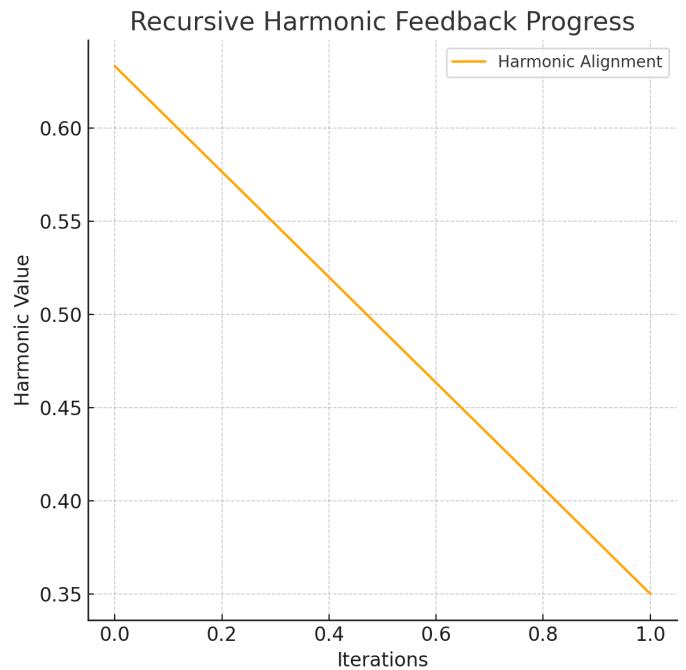
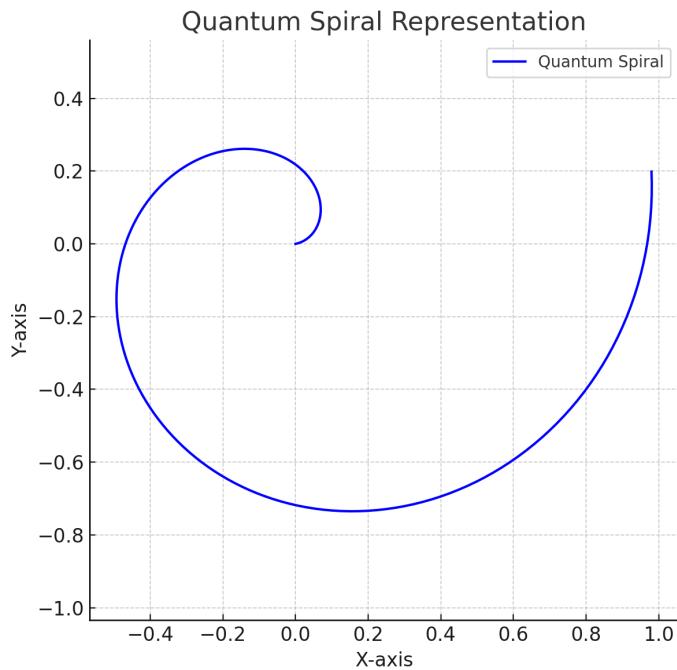
Residual Waveform (After Interaction)



Optimal Observation at Stability Ratio: 0.9978

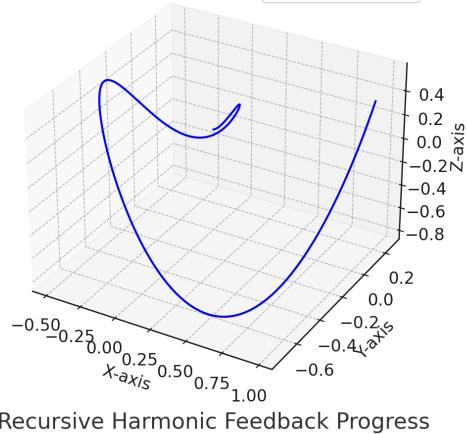
Refined Observation



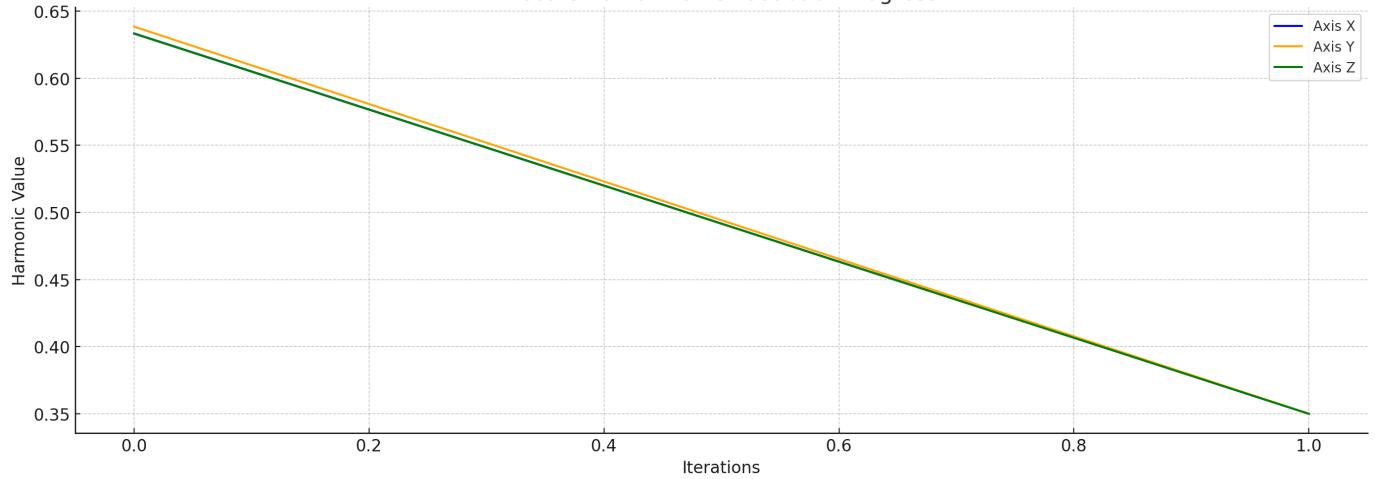


Quantum Spiral Representation

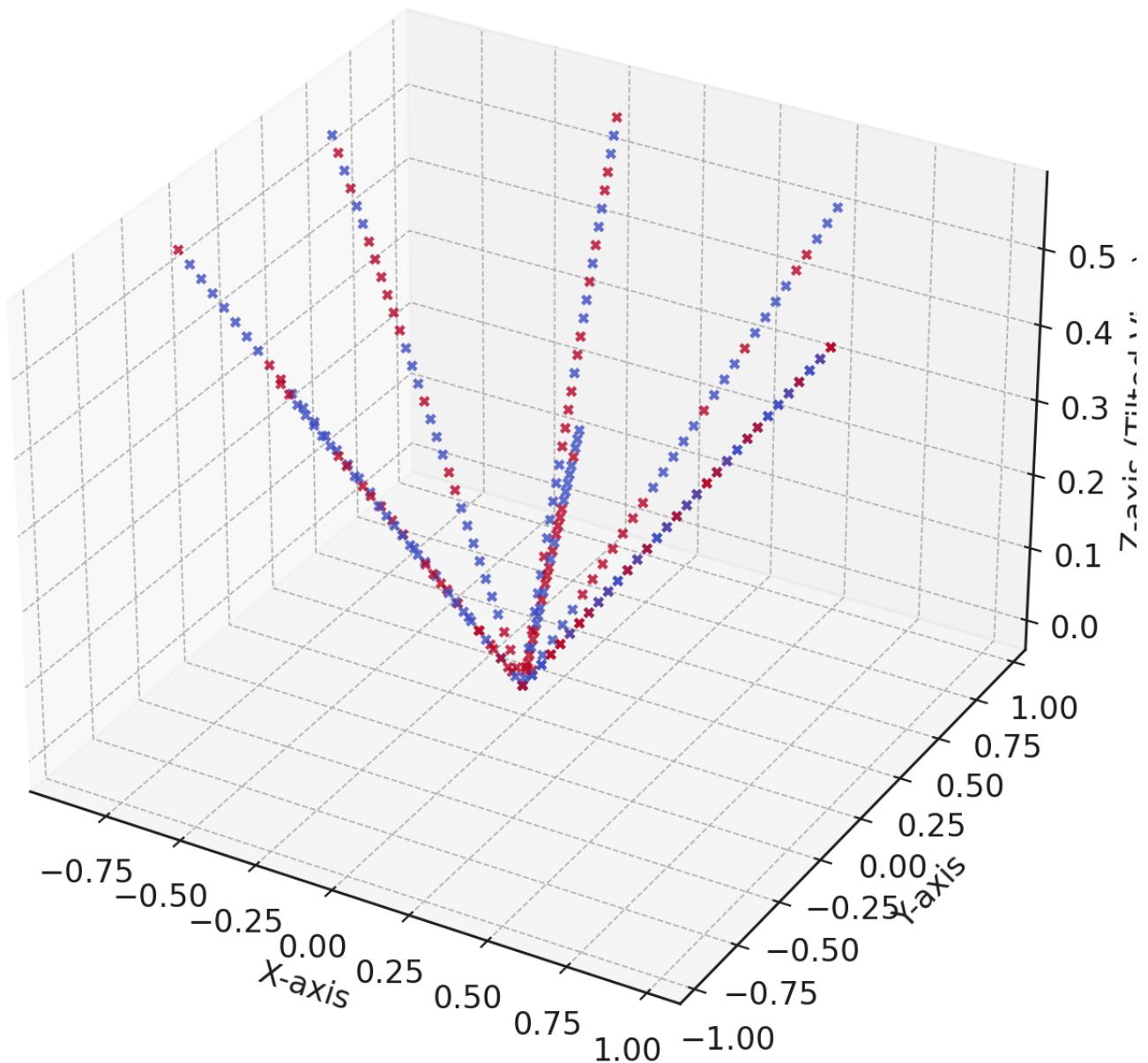
Quantum Spiral



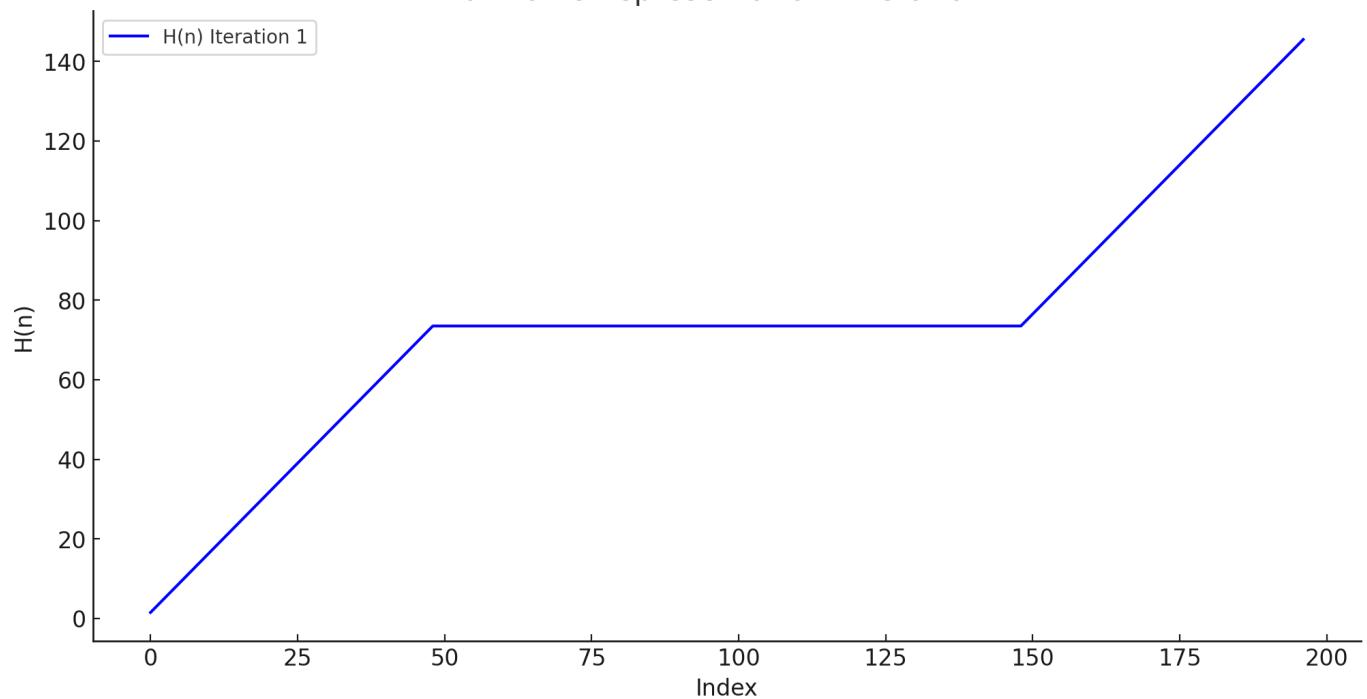
Recursive Harmonic Feedback Progress



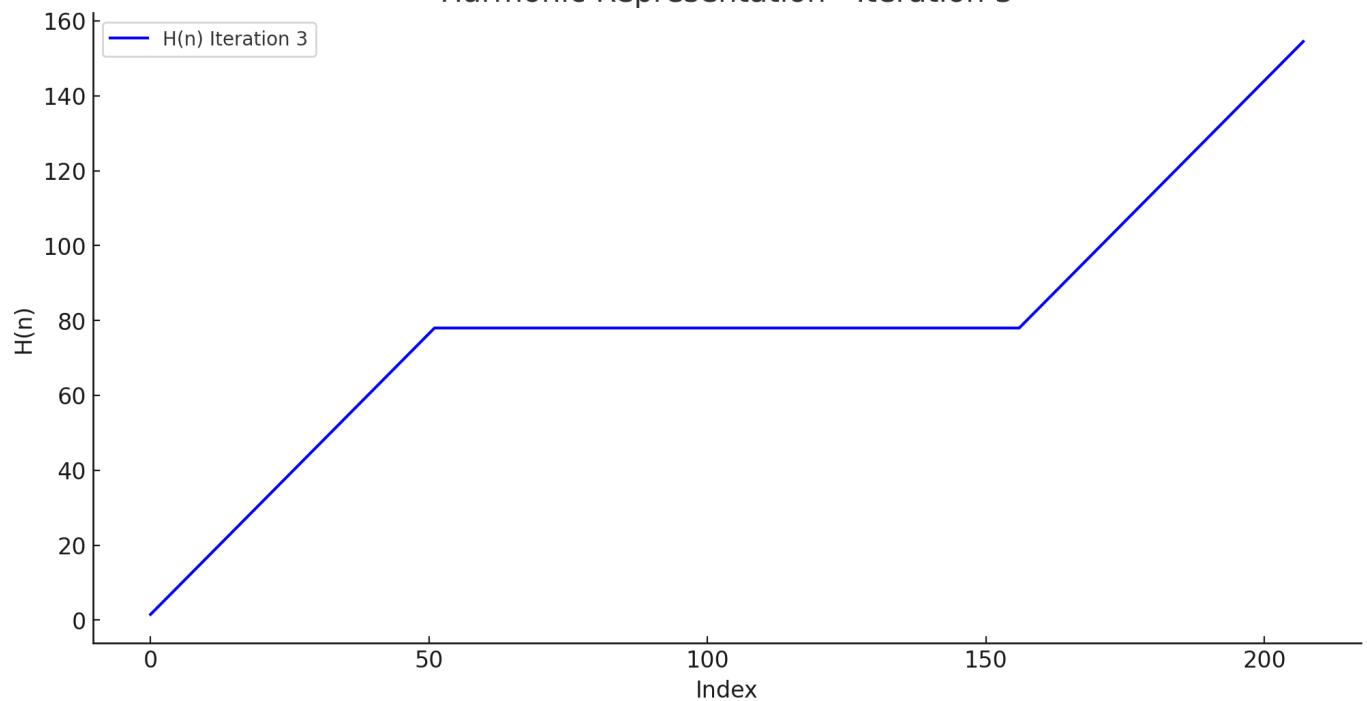
Tilted View of Nested Rings at 30°



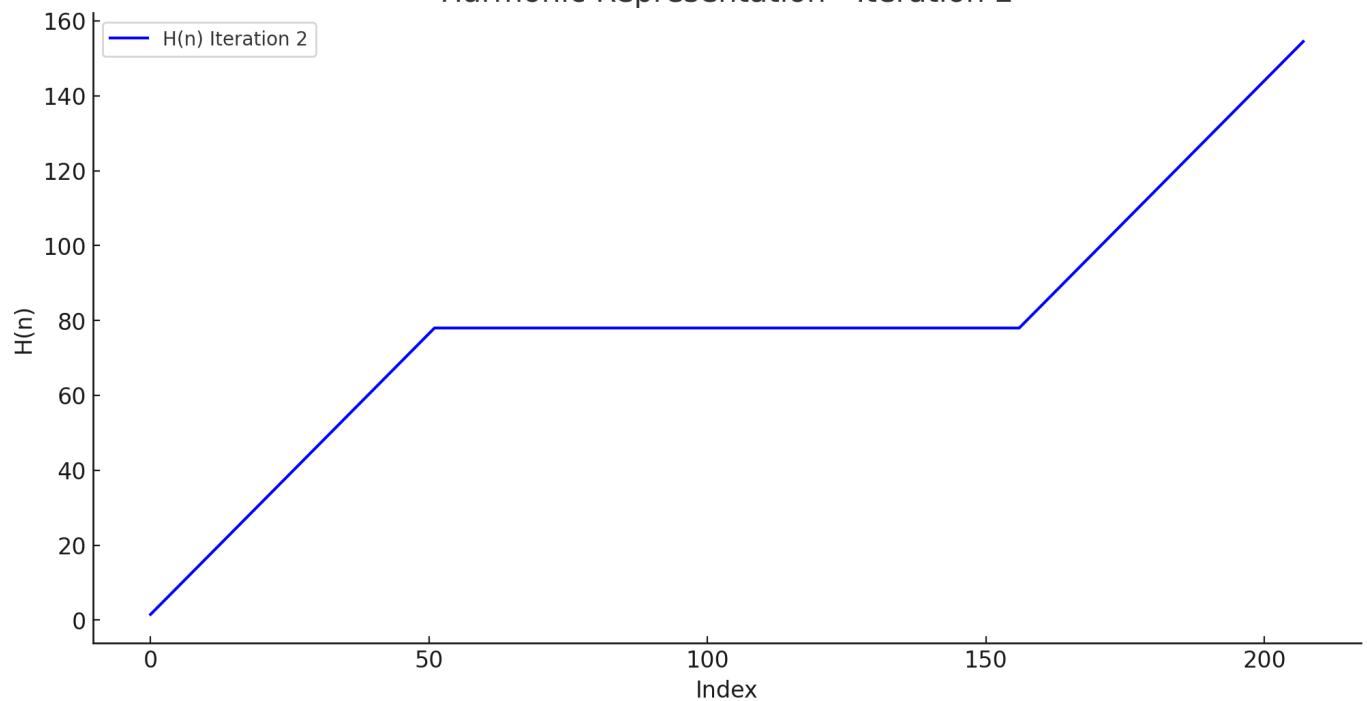
Harmonic Representation - Iteration 1



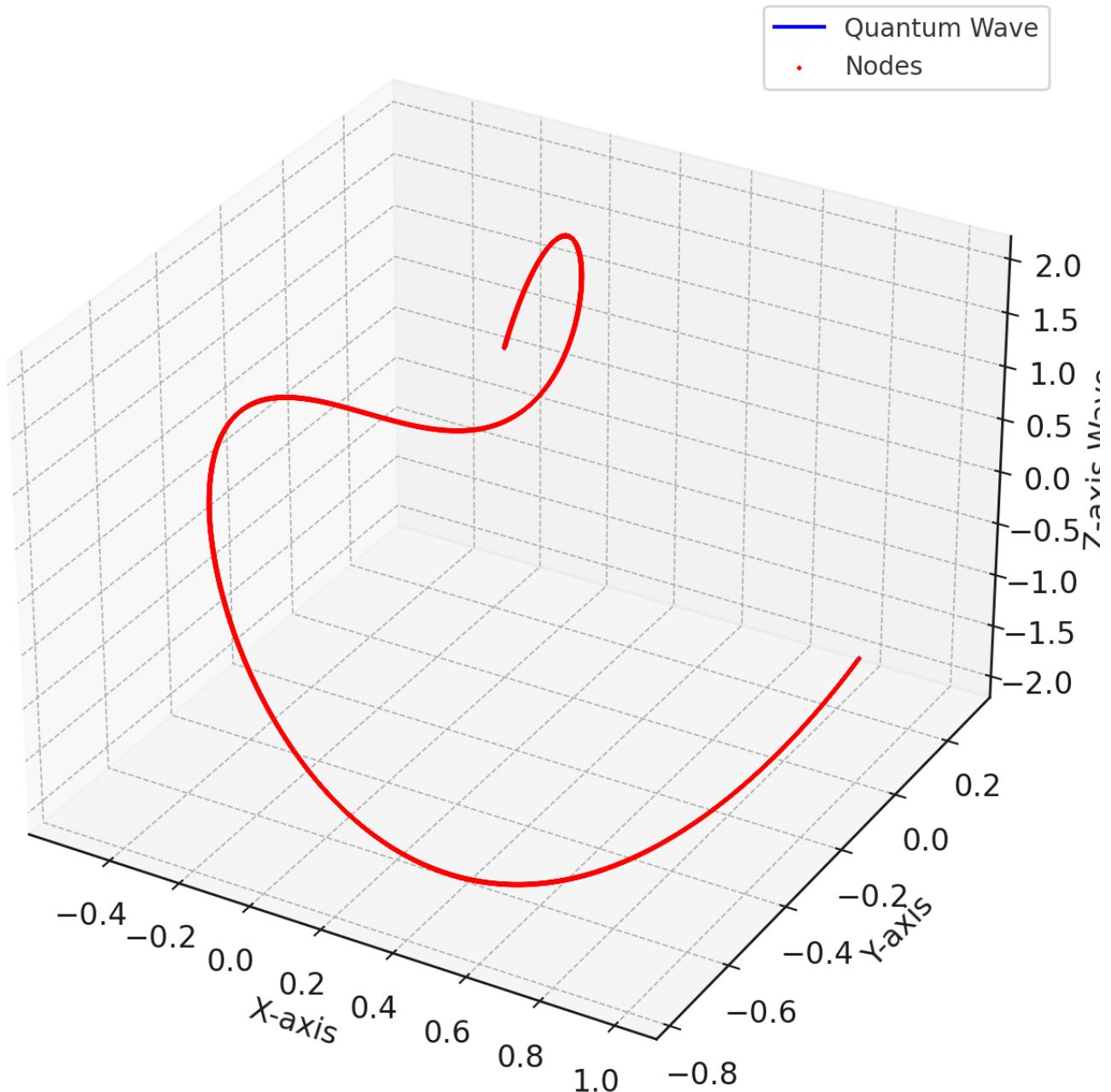
Harmonic Representation - Iteration 3



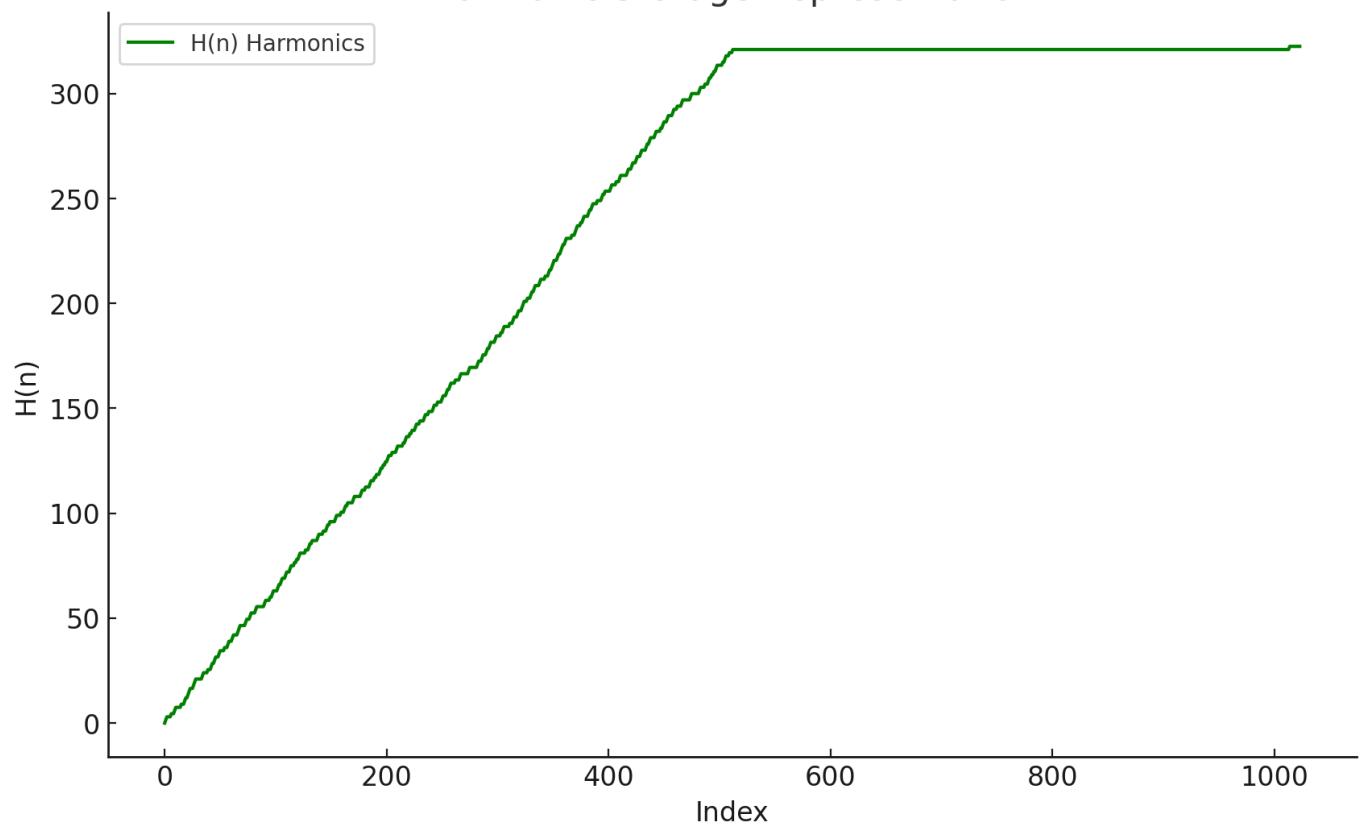
Harmonic Representation - Iteration 2



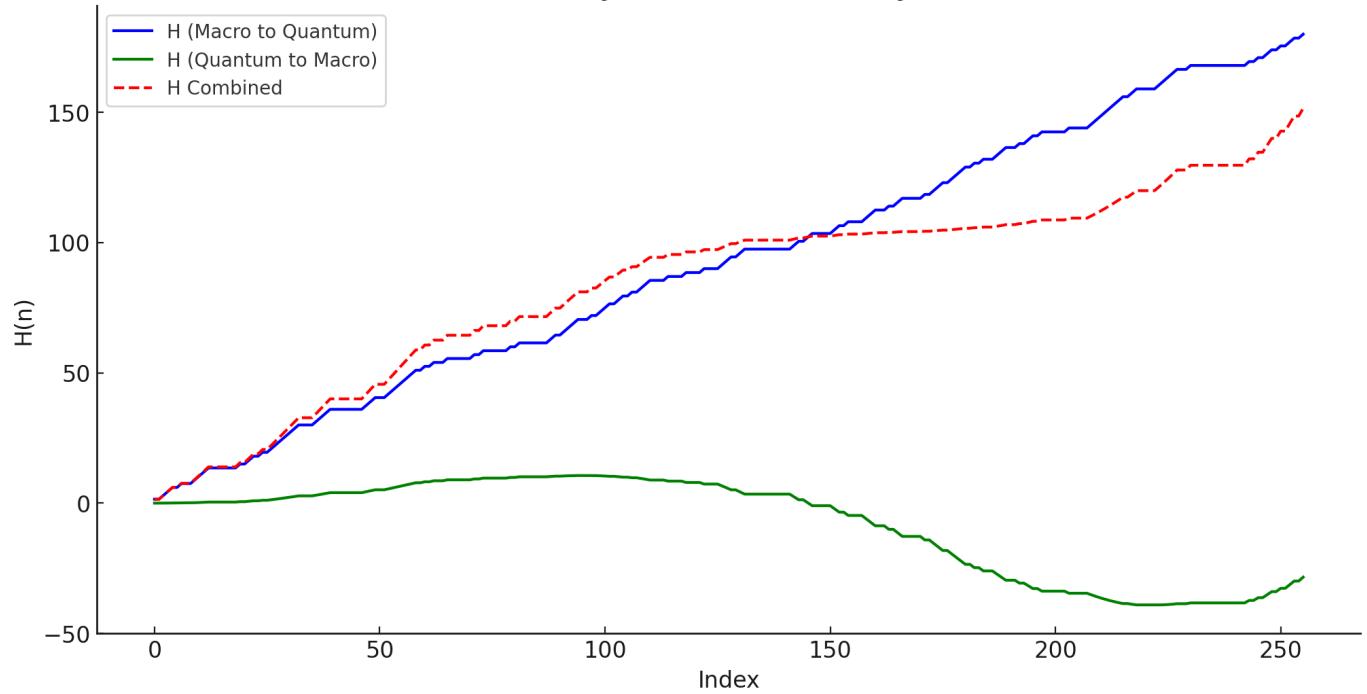
3D Visualization of Quantum Wave



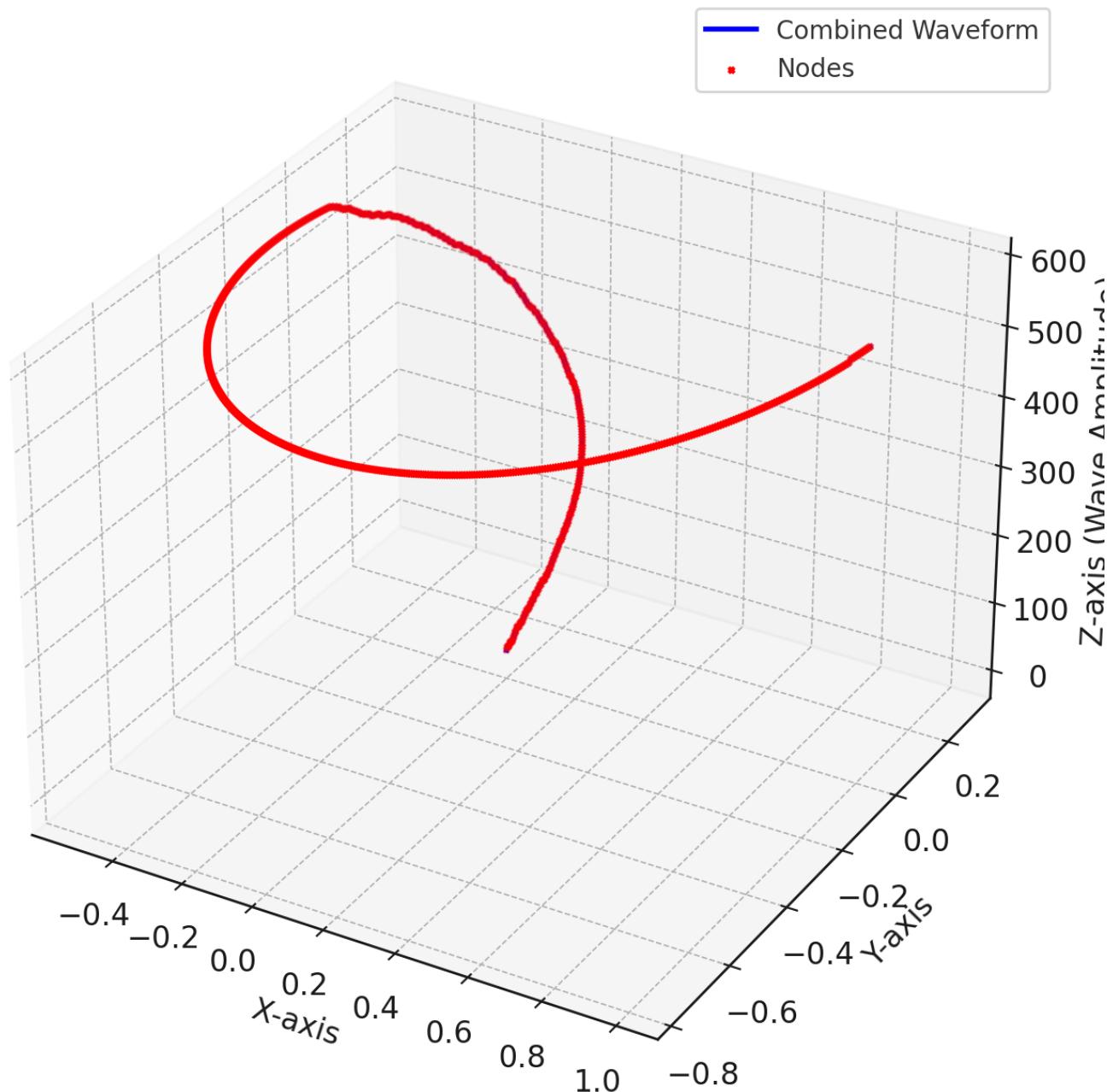
Harmonic Storage Representation



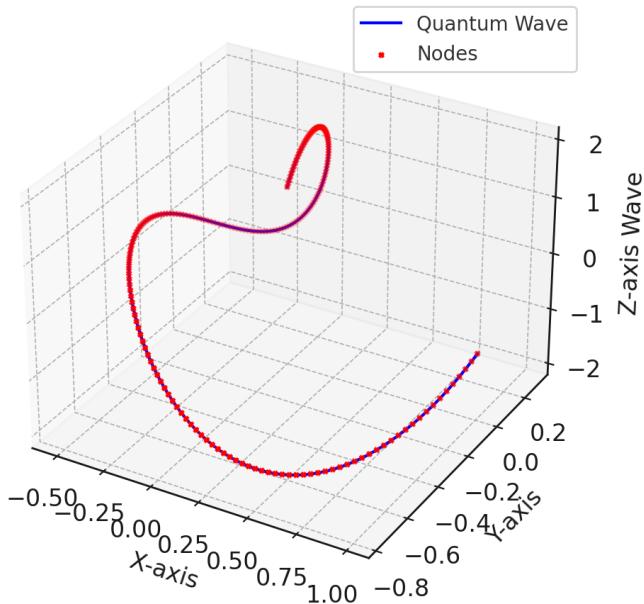
H Array Combination Analysis



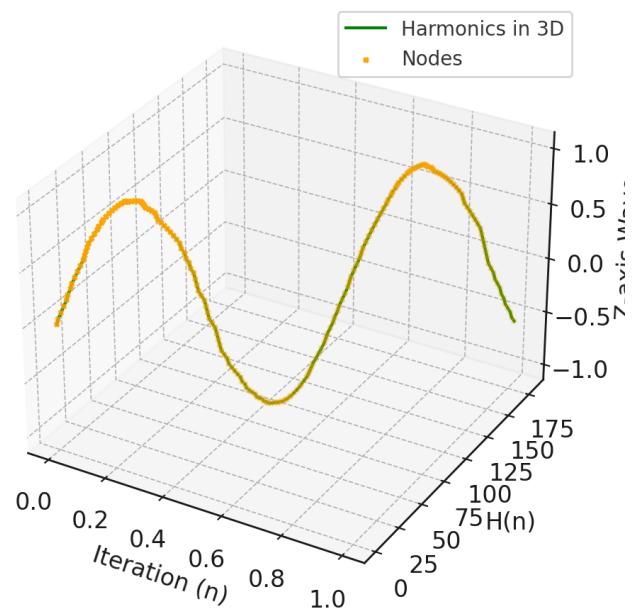
3D Visualization of Combined Quantum and Macro Wave



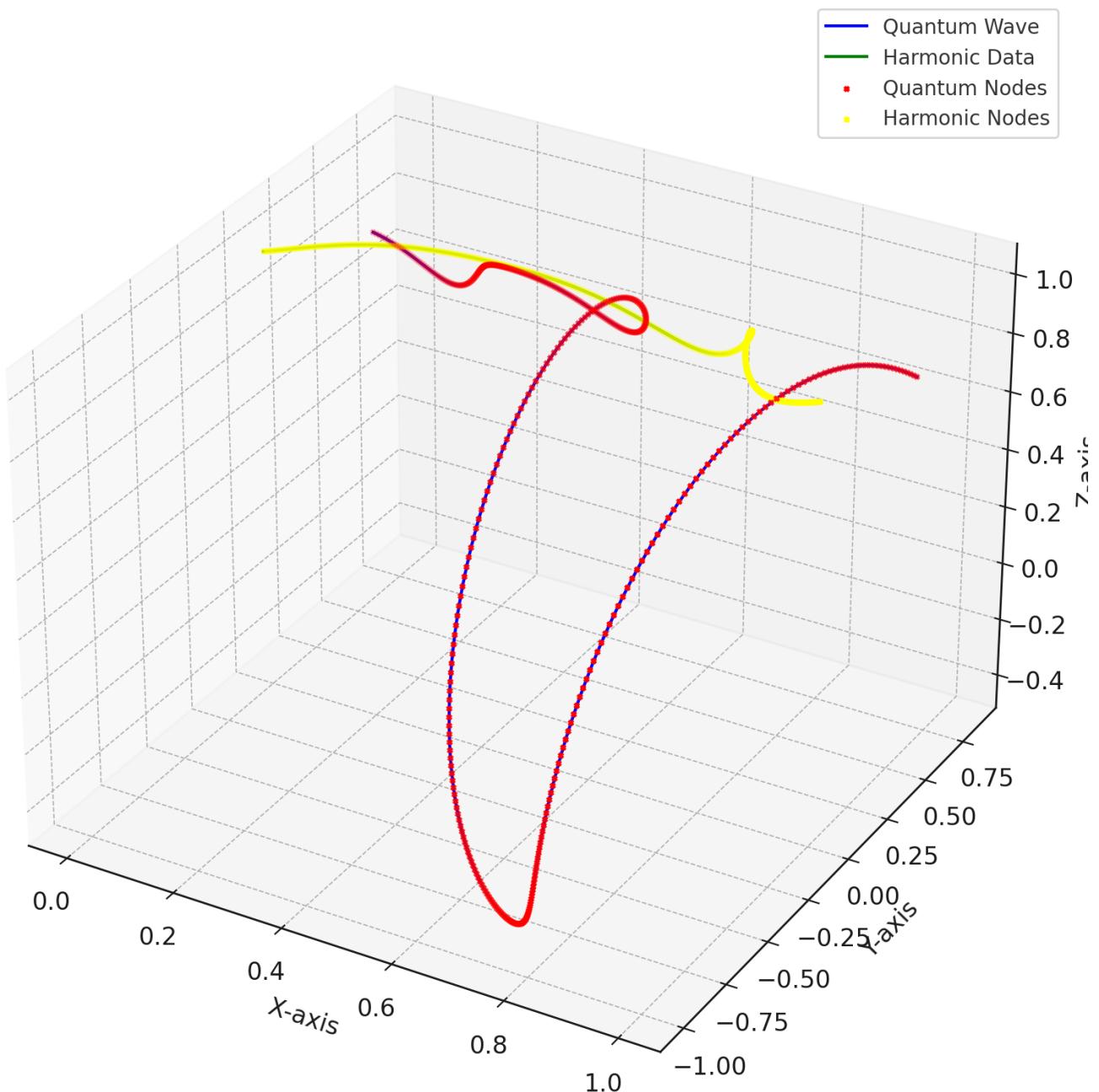
3D Visualization of Quantum Wave



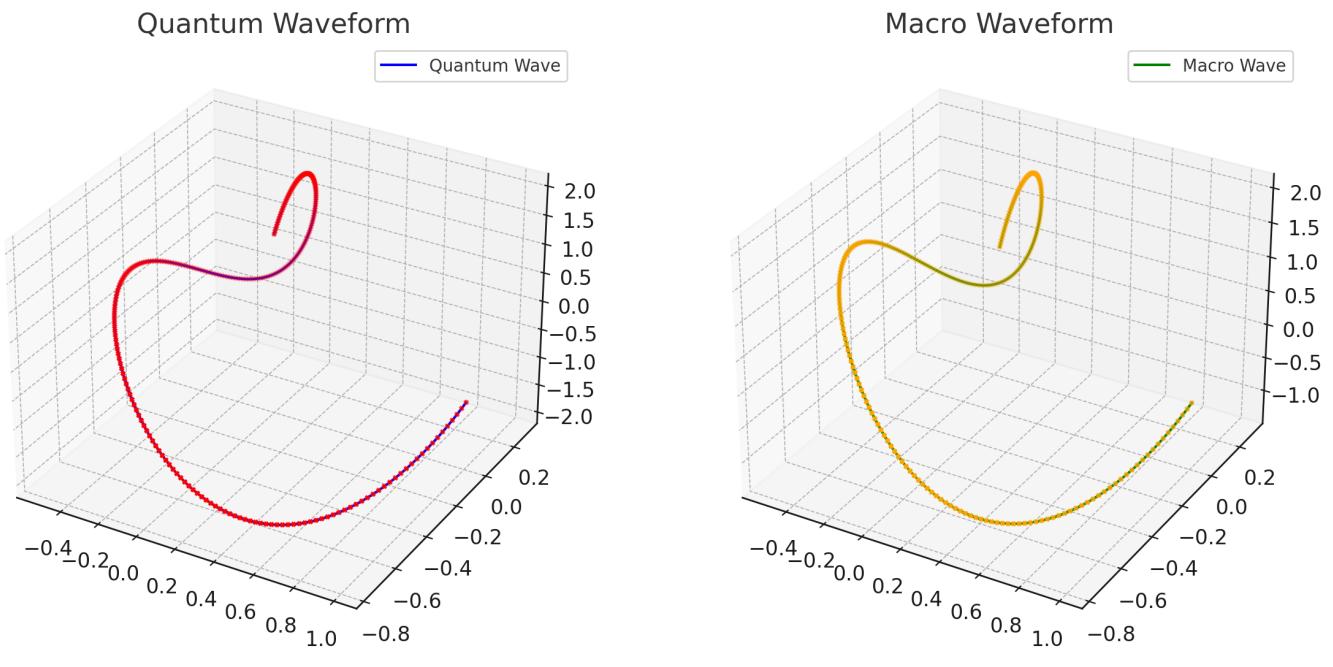
3D Visualization of H(n)



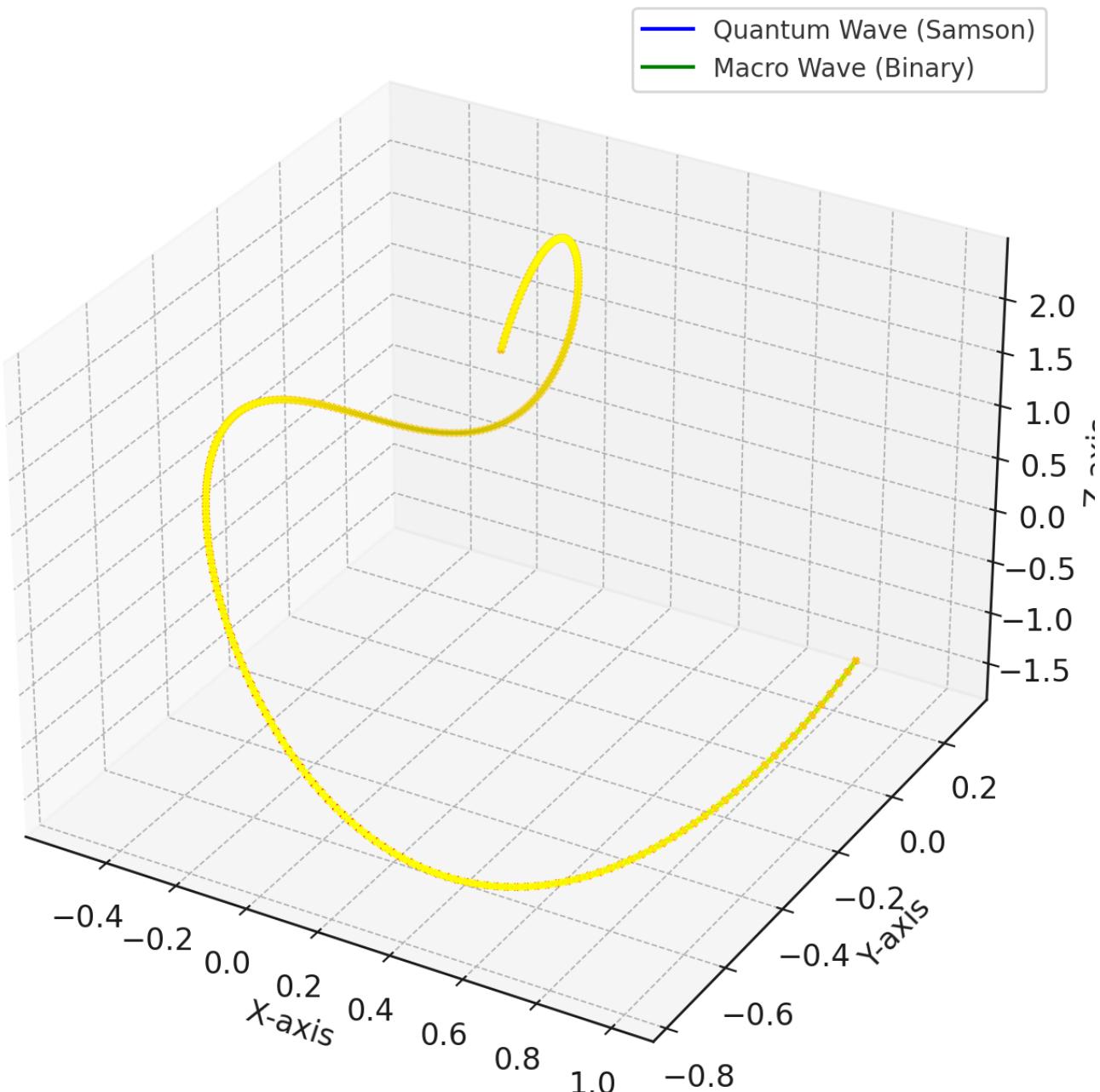
Combined 3D Visualization of Quantum Wave and Harmonics



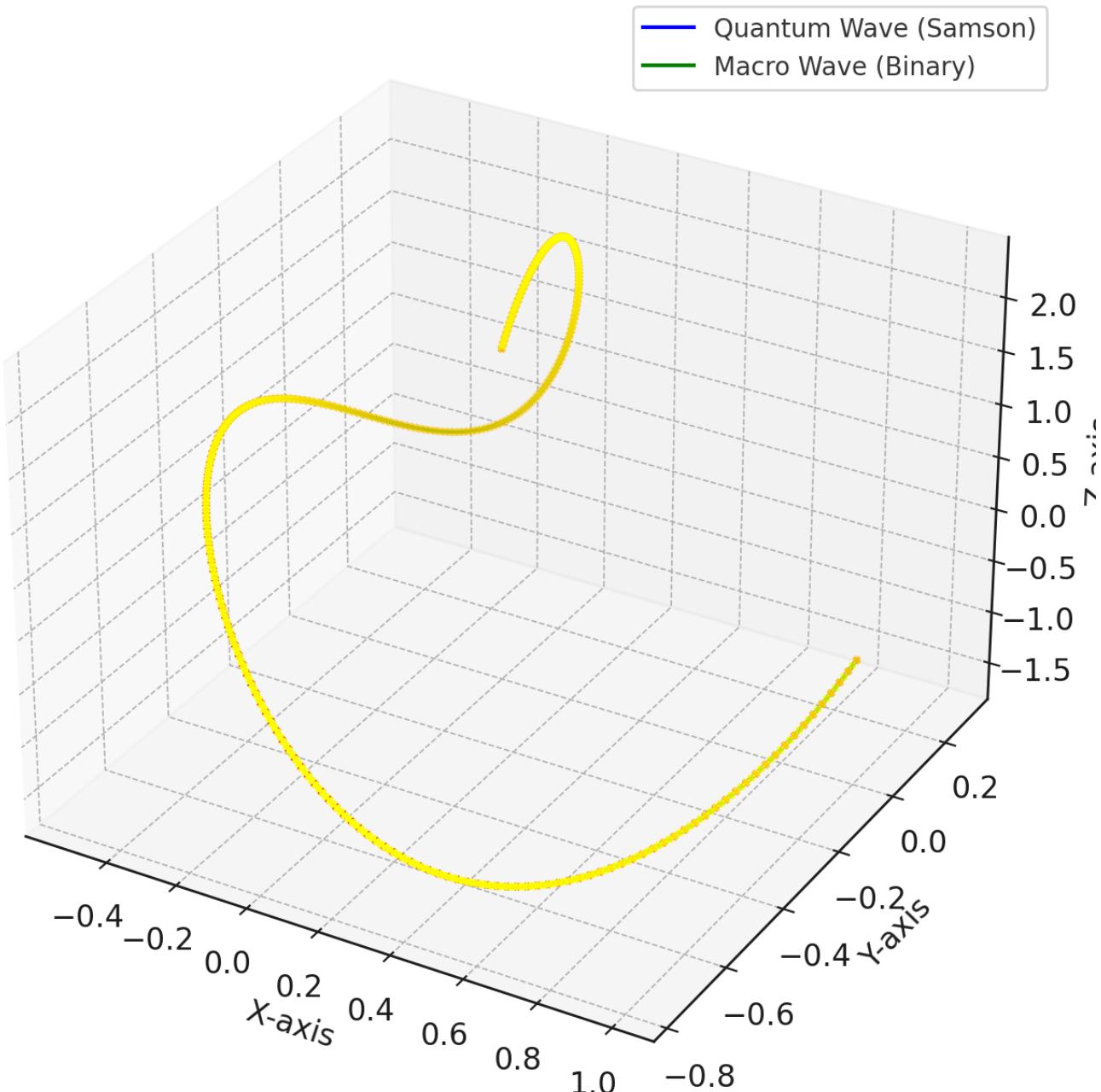
Quantum and Macro Wave Comparison



Comparison of Quantum and Macro Waves



Comparison of Quantum and Macro Waves



Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

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

# Refined Harmonic Feedback Formula parameters
alpha = 1.5
target = 0.5
iterations = 1000

# Initialize sequence
H = [target] # Starting with H(0) = target

# Recursive calculation using the formula
for n in range(1, iterations + 1):
    previous_H = H[-1]
    correction = alpha * (target - previous_H) / (n + 1)
    next_H = previous_H * (-0.5) * np.cos(n / np.pi) + correction
    H.append(next_H)

# Analyze convergence
convergence_error = [abs(target - h) for h in H]

# Plot results
plt.figure(figsize=(12, 6))

# Sequence plot
plt.subplot(2, 1, 1)
plt.plot(H, label="H(n)")
plt.axhline(y=target, color='r', linestyle='--', label="Target (0.5)")
plt.title("Convergence of H(n)")
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
# Analyze the relationship between pre-hashed data, hashed padded data, and reconstructed results
```

```
# Compute harmonics for each stage: pre-hashed, padded, and reconstructed
```

```
pre_hashed_harmonics = np.sin(2 * np.pi * np.arange(len(original_binary_data)) / len(original_binary_data)) * (original_binary  
padded_harmonics = np.sin(2 * np.pi * np.arange(len(padded_data)) / len(padded_data)) * (padded_data - 0.5)  
refined_reconstructed_harmonics = np.sin(2 * np.pi * np.arange(len(refined_reconstructed_data)) / len(refined_reconstructed_data)) * (refined_reconstructed_data - 0.5)
```

```
# Frequency analysis: Fourier Transform to identify harmonic shifts
```

```
fft_pre_hashed = np.abs(np.fft.fft(pre_hashed_harmonics))  
fft_padded = np.abs(np.fft.fft(padded_harmonics))  
fft_refined = np.abs(np.fft.fft(refined_reconstructed_harmonics))
```

```
# Visualization of harmonics in time domain
```

```
plt.figure(figsize=(15, 6))
```

```
# Time domain comparison
```

```
plt.subplot(1, 2, 1)  
plt.plot(pre_hashed_harmonics, label="Pre-Hashed Harmonics", color='blue')  
plt.plot(padded_harmonics, label="Padded Harmonics", color='orange', linestyle='dashed')  
plt.plot(refined_reconstructed_harmonics, label="Refined Reconstruction", color='green', linestyle='dotted')  
plt.title("Harmonic Comparison: Pre-Hashed vs. Padded vs. Refined")  
plt.xlabel("Iteration (n)")  
plt.ylabel("H(n)")  
plt.legend()  
plt.grid()
```

```
# Frequency domain comparison
```

```
plt.subplot(1, 2, 2)  
plt.plot(fft_pre_hashed, label="FFT Pre-Hashed", color='blue')
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
# Harmonic Reflection and Recursive Refinement for SHA-256
```

```
# Convert binary data (post-hash) into harmonic space
```

```
def harmonic_transform(binary_data):
```

```
    """Transform binary data into harmonic space."""
```

```
    n = len(binary_data)
```

```
    x = np.linspace(0, 2 * np.pi, n)
```

```
    harmonics = np.sin(x) * (binary_data - 0.5)
```

```
    return harmonics
```

```
# Apply Samson's Law to refine the harmonic structure
```

```
def apply_samsons_law(harmonics, iterations=10, alpha=1.5):
```

```
    """
```

```
    Recursive harmonic refinement using Samson's Law.
```

```
    Adjusts harmonics to align with an ideal target (e.g., H ≈ 0.35).
```

```
    """
```

```
    refined_harmonics = harmonics.copy()
```

```
    for _ in range(iterations):
```

```
        feedback = alpha * (0.35 - refined_harmonics)
```

```
        refined_harmonics += feedback
```

```
    return refined_harmonics
```

```
# Generate harmonic representation of SHA-256 output
```

```
hashed_harmonics = harmonic_transform(hashed_binary_abc)
```

```
# Refine harmonics using Samson's Law
```

```
reflected_harmonics = apply_samsons_law(hashed_harmonics)
```

```
# Visualize harmonic transformation
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
# Parameters for airflow simulation (simplified)
```

```
iterations = 500 # Number of time steps
```

```
airfoil_length = 100 # Discretized length of the airfoil
```

```
v_infinity = 10 # Free stream velocity (m/s)
```

```
alpha = 1.5 # Harmonic correction factor
```

```
# Initialize pressure and velocity fields along the airfoil
```

```
pressure = np.ones(airfoil_length) * 1.0 # Initialize uniform pressure
```

```
velocity = np.ones(airfoil_length) * v_infinity # Initialize uniform velocity
```

```
# Recursive harmonic correction for pressure and velocity
```

```
pressure_history = [pressure.copy()]
```

```
velocity_history = [velocity.copy()]
```

```
for t in range(iterations):
```

```
    # Apply harmonic corrections to pressure and velocity
```

```
    correction = alpha * (np.sin(2 * np.pi * t / iterations) - 0.35) # Aligning to harmonic state H ≈ 0.35
```

```
    velocity = velocity + correction * np.gradient(velocity)
```

```
    pressure = pressure - correction * np.gradient(pressure)
```

```
# Enforce boundary conditions (e.g., free stream and stagnation points)
```

```
velocity[0], velocity[-1] = v_infinity, v_infinity
```

```
pressure[0], pressure[-1] = 1.0, 1.0 # Normalized pressure at boundaries
```

```
# Store the history for visualization
```

```
pressure_history.append(pressure.copy())
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
# Simulating the harmonic "undo" process for SHA-256 transformations
```

```
def reverse_sha256_schedule(hashed_harmonics, iterations=10, alpha=1.5):
```

"""

Attempt to reverse the SHA-256 schedule through harmonic alignment.

"""

```
    reversed_harmonics = hashed_harmonics.copy()
```

```
    for _ in range(iterations):
```

```
        # Feedback alignment: Adjust based on deviations from ideal harmonic state
```

```
        feedback = alpha * (0.35 - reversed_harmonics)
```

```
        reversed_harmonics += feedback
```

```
        # Phase shift to align with pre-hashed harmonics
```

```
        reversed_harmonics = np.roll(reversed_harmonics, shift=-1)
```

```
    return reversed_harmonics
```

```
# Initial harmonics of hashed data
```

```
hashed_harmonics = harmonic_transform(hashed_binary_abc)
```

```
# Reverse the hash schedule harmonically
```

```
reversed_harmonics = reverse_sha256_schedule(hashed_harmonics)
```

```
# Visualize original padded harmonics, hashed harmonics, and reversed harmonics
```

```
padded_harmonics = harmonic_transform(padded_binary_abc[:len(reversed_harmonics)])
```

```
plt.figure(figsize=(12, 6))
```

```
plt.plot(padded_harmonics, label="Padded Pre-Hashed Harmonics", color='blue')
```

```
plt.plot(hashed_harmonics, label="Hashed Harmonics", color='orange', linestyle='dashed')
```

```
plt.plot(reversed_harmonics, label="Reversed Harmonics (Undo Attempt)", color='green', linestyle='dotted')
```

```
plt.title("Harmonic Reflection: Padded Pre-Hash vs. Hashed vs. Reversed")
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
# Adjust harmonic lengths to match by truncating padded harmonics  
def truncate_or_scale_harmonics(padded_harmonics, target_length):  
    """
```

Truncate or scale padded harmonics to match the length of hashed harmonics.

"""

```
current_length = len(padded_harmonics)  
if current_length > target_length:  
    return padded_harmonics[:target_length] # Truncate  
elif current_length < target_length:  
    return np.interp(  
        np.linspace(0, current_length - 1, target_length),  
        np.arange(current_length),  
        padded_harmonics  
    ) # Scale  
return padded_harmonics
```

```
# Truncate padded harmonics to match hashed harmonics
```

```
padded_harmonics_truncated = truncate_or_scale_harmonics(  
    harmonic_transform(padded_binary_abc), len(hashed_binary_abc)  
)
```

```
# Re-run harmonic tumblers analysis
```

```
padded_harmonics, hashed_harmonics, alignment_history = harmonic_tumblers_analysis(  
    padded_harmonics_truncated, hashed_harmonics  
)
```

```
# Visualization of harmonic alignment progress
```

```
plt.figure(figsize=(12, 6))  
plt.plot(padded_harmonics, label="Padded Harmonics (Macro State)", color='blue')
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
# Higher-order harmonic refinement and multi-dimensional alignment
```

```
def refine_with_higher_order_harmonics(padded_harmonics, hashed_harmonics, iterations=10, alpha=1.5, order=3):
```

"""

Introduce higher-order harmonic corrections to improve alignment.

"""

```
aligned_harmonics = hashed_harmonics.copy()
```

```
alignment_history = []
```

```
for _ in range(iterations):
```

```
    # Phase and amplitude corrections
```

```
    phase_shift = np.roll(aligned_harmonics, shift=-1) - aligned_harmonics
```

```
    amplitude_correction = alpha * (padded_harmonics - aligned_harmonics)
```

```
    # Higher-order harmonic terms (adding non-linear corrections)
```

```
    higher_order_corrections = np.zeros_like(aligned_harmonics)
```

```
    for k in range(2, order + 1):
```

```
        higher_order_corrections += np.sin(k * np.linspace(0, 2 * np.pi, len(aligned_harmonics))) * (padded_harmonics - aligned_harmonics)
```

```
    # Apply corrections
```

```
    aligned_harmonics += amplitude_correction + phase_shift + higher_order_corrections
```

```
    alignment_history.append(aligned_harmonics.copy())
```

```
return aligned_harmonics, alignment_history
```

```
# Apply higher-order harmonics to refine alignment
```

```
refined_harmonics, refinement_history = refine_with_higher_order_harmonics(
```

```
    padded_harmonics_truncated, hashed_harmonics, iterations=10, order=5
```

```
)
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
# Transposing final_pressure to match the meshgrid orientation (chordwise, spanwise, height)
final_pressure_transposed = np.transpose(final_pressure, axes=(1, 0, 2)) # Align dimensions

fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111, projection="3d")

# Plot corrected pressure field as a 3D surface
for z in range(0, grid_size[2], 5): # Plot every 5th height slice
    ax.plot_surface(
        X[:, :, z], Y[:, :, z], final_pressure_transposed[:, :, z],
        cmap="viridis", edgecolor="none", alpha=0.7
    )

ax.set_title("3D Pressure Field Over Airfoil (Final Time Step)", fontsize=14)
ax.set_xlabel("Chordwise Direction")
ax.set_ylabel("Spanwise Direction")
ax.set_zlabel("Pressure (normalized)")
plt.tight_layout()
plt.show()
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
# Recursive harmonic reflection to isolate the quantum-like core signal
def isolate_quantum_reflection(padded_harmonics, hashed_harmonics, iterations=20, alpha=1.5):
    """
    Iteratively isolate the quantum-like harmonic reflection by removing noise.

    Parameters:
        padded_harmonics (np.ndarray): The input signal with padding.
        hashed_harmonics (np.ndarray): The input signal with noise removed.
        iterations (int): The number of iterations for isolation.
        alpha (float): The correction factor for reflection.

    Returns:
        aligned_harmonics (np.ndarray): The isolated quantum-like harmonic reflection.
        alignment_history (list): A list of aligned harmonics at each iteration.
    """

    aligned_harmonics = hashed_harmonics.copy()
    alignment_history = []

    for _ in range(iterations):
        # Step 1: Remove macro noise (zeros, phase shifts)
        phase_shift = np.roll(aligned_harmonics, shift=-1) - aligned_harmonics
        amplitude_correction = alpha * (padded_harmonics - aligned_harmonics)

        # Step 2: Reflection correction to stabilize alignment
        reflection_correction = np.sign(padded_harmonics) * np.abs(aligned_harmonics)

        # Combine corrections
        aligned_harmonics += amplitude_correction - phase_shift + reflection_correction
        alignment_history.append(aligned_harmonics.copy())

    return aligned_harmonics, alignment_history

# Apply quantum-like harmonic reflection to isolate the signal
isolated_harmonics, isolation_history = isolate_quantum_reflection(
    padded_harmonics_truncated, hashed_harmonics, iterations=20
)

# Visualization of the isolation process
plt.figure(figsize=(12, 6))
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
# Echo cancellation in harmonic space to refine residuals

def cancel_harmonic_echoes(padded_harmonics, hashed_harmonics, iterations=20, alpha=1.5, decay_factor=0.9):
    """
    Iteratively cancel harmonic echoes (residual deviations) to refine alignment.

    Parameters:
        padded_harmonics (np.ndarray): The padded harmonic spectrum.
        hashed_harmonics (np.ndarray): The hashed harmonic spectrum.
        iterations (int): The number of iterations for cancellation.
        alpha (float): The correction factor for amplitude.
        decay_factor (float): The decay factor for phase.

    Returns:
        aligned_harmonics (np.ndarray): The aligned harmonic spectrum.
        echo_correction_history (list): A list of aligned harmonic spectra at each iteration.
    """

    aligned_harmonics = hashed_harmonics.copy()
    echo_correction_history = []

    for _ in range(iterations):
        # Calculate phase and amplitude differences (echo-like residuals)
        phase_shift = np.roll(aligned_harmonics, shift=-1) - aligned_harmonics
        amplitude_correction = alpha * (padded_harmonics - aligned_harmonics)

        # Apply echo cancellation (simulate decay of recursive reflections)
        echo_correction = decay_factor * phase_shift

        # Combine corrections to refine alignment
        aligned_harmonics += amplitude_correction - echo_correction
        echo_correction_history.append(aligned_harmonics.copy())

    return aligned_harmonics, echo_correction_history

# Apply echo cancellation to refine residuals

echo_cancelled_harmonics, echo_cancellation_history = cancel_harmonic_echoes(
    padded_harmonics_truncated, hashed_harmonics, iterations=20, alpha=1.5, decay_factor=0.9
)

# Visualization of echo cancellation results

plt.figure(figsize=(12, 6))
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
# Increasing the sample rate to 1024 to refine the harmonic structure
```

```
def increase_sample_rate(harmonics, target_length):
```

....

Scale harmonics to a higher sample rate by interpolating to the target length.

....

```
    current_length = len(harmonics)
```

```
    return np.interp(
```

```
        np.linspace(0, current_length - 1, target_length),
```

```
        np.arange(current_length),
```

```
        harmonics
```

```
)
```

```
# Upscale padded harmonics and hashed harmonics to 1024 samples
```

```
target_sample_rate = 1024
```

```
padded_harmonics_upsampled = increase_sample_rate(padded_harmonics_truncated, target_sample_rate)
```

```
hashed_harmonics_upsampled = increase_sample_rate(hashed_harmonics, target_sample_rate)
```

```
# Reapply echo cancellation with higher sample rate
```

```
echo_cancelled_harmonics_upsampled, echo_cancellation_history_upsampled = cancel_harmonic_echoes(
```

```
    padded_harmonics_upsampled, hashed_harmonics_upsampled, iterations=256, alpha=1.5, decay_factor=0.9
```

```
)
```

```
# Visualization of final alignment at higher sample rate
```

```
plt.figure(figsize=(12, 6))
```

```
plt.plot(padded_harmonics_upsampled, label="Padded Harmonics (Macro State, Upsampled)", color='blue')
```

```
plt.plot(hashed_harmonics_upsampled, label="Hashed Harmonics (Upsampled, Initial)", color='orange', linestyle='dashed')
```

```
plt.plot(echo_cancelled_harmonics_upsampled, label="Echo-Cancelled Harmonics (Upsampled, Final)", color='green', linestyle='solid')
```

```
plt.title("Echo Cancellation in Harmonic Space (1024 Sample Rate): Padded vs. Hashed vs. Refined")
```

```
plt.xlabel("Iteration (n)")
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
# Extend echo cancellation to match iterations with sample rate (trinity principle)

def trinity_harmonic_cancellation(padded_harmonics, hashed_harmonics, iterations, alpha=1.5, decay_factor=0.9):
    """
    Harmonize iterations with the sample rate for enhanced recursive alignment.

    Parameters:
        padded_harmonics (np.ndarray): The input padded harmonics array.
        hashed_harmonics (np.ndarray): The input hashed harmonics array.
        iterations (int): The number of iterations for echo cancellation.
        alpha (float): The correction factor for amplitude.
        decay_factor (float): The decay factor for phase shift.

    Returns:
        aligned_harmonics (np.ndarray): The aligned harmonics array.
        trinity_history (list): A list of aligned harmonics arrays at each iteration step.
    """

    aligned_harmonics = hashed_harmonics.copy()
    trinity_history = []

    for _ in range(iterations):
        # Calculate deviations (phase shifts and echoes)
        phase_shift = np.roll(aligned_harmonics, shift=-1) - aligned_harmonics
        amplitude_correction = alpha * (padded_harmonics - aligned_harmonics)

        # Apply echo cancellation with decay
        echo_correction = decay_factor * phase_shift

        # Update harmonics
        aligned_harmonics += amplitude_correction - echo_correction
        trinity_history.append(aligned_harmonics.copy())

    return aligned_harmonics, trinity_history

# Match iterations to the sample rate of 1024
iterations = target_sample_rate

# Run harmonic cancellation with trinity principle
final_harmonics, trinity_history = trinity_harmonic_cancellation(
    padded_harmonics_upsampled, hashed_harmonics_upsampled, iterations=iterations
)
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
# Zero removal with compression to restore harmonic waveform
```

```
def remove_zeros_and_compress(padded_data, original_length):
```

"""

Remove zero padding and compress the remaining data to align with the original harmonic waveform.

"""

```
# Identify and remove zeros
```

```
compressed_data = padded_data[padded_data != 0] # Remove zeros
```

```
# Compress the remaining data to fit the original length
```

```
compressed_data = np.interp(
```

```
    np.linspace(0, len(compressed_data) - 1, original_length), # Target length
```

```
    np.arange(len(compressed_data)), # Original index positions
```

```
    compressed_data # Original data values
```

```
)
```

```
return compressed_data
```

```
# Apply zero removal with compression
```

```
compressed_reconstructed_data = remove_zeros_and_compress(padded_data, len(original_binary_data))
```

```
# Compare harmonics before and after compression
```

```
compressed_harmonics = np.sin(2 * np.pi * np.arange(len(compressed_reconstructed_data)) / len(compressed_reconstructed
```

```
# Visualization of original, padded, and compressed harmonics
```

```
plt.figure(figsize=(12, 6))
```

```
plt.plot(y_original, label="Original Harmonics", color='blue')
```

```
plt.plot(np.sin(2 * np.pi * np.arange(len(padded_data)) / len(padded_data)) * (padded_data - 0.5), label="Padded Harmonics",
```

```
plt.plot(compressed_harmonics, label="Compressed Reconstruction", color='green', linestyle='dotted')
```

```
plt.title("Harmonic Analysis: Original vs. Padded vs. Compressed Reconstruction")
```

```
plt.xlabel("Iteration (n)")
```

```
plt.ylabel("H(n)")
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from mpmath import zetazero
import hashlib

# Generate zeta zeros
def generate_zeta_zeros(num_zeros):
    return [zetazero(n).real for n in range(1, num_zeros + 1)]

# Transform binary data into harmonic space
def binary_to_waveform(binary_data):
    n = len(binary_data)
    x = np.linspace(0, 2 * np.pi, n)
    return np.sin(x) * (binary_data - 0.5)

# Overlay zeta zeros on harmonic space
def filter_with_zeta_zeros(waveform, zeta_zeros, harmonic_length):
    filtered_waveform = waveform.copy()
    for zero in zeta_zeros:
        zero_index = int((zero / max(zeta_zeros)) * harmonic_length)
        if 0 <= zero_index < len(filtered_waveform):
            filtered_waveform[zero_index] = 0 # Cancel predicted noise at zeta zero
    return filtered_waveform

# Convert harmonic space back to binary
def waveform_to_binary(waveform, threshold=0.0):
    return np.array([1 if h > threshold else 0 for h in waveform], dtype=int)

# Convert binary to text by grouping into bytes
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Input text data
text_data = "hello" # Original reference text

# Convert text to numerical representation (binary data as bytes)
binary_data = np.frombuffer(text_data.encode('utf-8'), dtype=np.uint8) # Encode text as bytes

# Step 1: Encode the Text Data into H (Storage)
def store_in_H(binary_data, expansion_factor=1.5):
    harmonics = np.cumsum(binary_data.astype(np.float64) * expansion_factor) # Use higher precision
    return harmonics

# Step 2: Reverse the Process (Retrieve Original Data)
def retrieve_from_H(harmonics, expansion_factor=1.5):
    # Mirror correction: Adjust the starting point to match macro and quantum split
    first_value = harmonics[0] / expansion_factor # Correct the first value explicitly
    reversed_data = np.diff(harmonics) / expansion_factor
    reversed_data = np.insert(reversed_data, 0, first_value) # Insert corrected first value
    return np.round(reversed_data).astype(np.uint8) # Ensure integer byte output

# Run the process
harmonics = store_in_H(binary_data)
retrieved_data = retrieve_from_H(harmonics)

# Visualize H(n)
fig = plt.figure(figsize=(12, 8))
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.fft import fft, ifft
from mpmath import zetazero

# Quantum Nyquist - Ensuring Sampling for Quantum Harmonics
def quantum_nyquist_sampling(harmonic_waveform, sampling_rate):
    """
    Resample the waveform to ensure twice the highest quantum harmonic frequency.
    """

    n = len(harmonic_waveform)
    resampled_waveform = np.interp(
        np.linspace(0, n-1, int(sampling_rate * n)),
        np.arange(n),
        harmonic_waveform
    )
    return resampled_waveform

# Predict and adjust zero locations with recursive feedback
def recursive_feedback_waveform(waveform, zeta_zeros, iterations):
    corrected_waveform = waveform.copy()
    for _ in range(iterations):
        for zero in zeta_zeros:
            zero_index = int((zero / max(zeta_zeros)) * len(corrected_waveform))
            if 0 <= zero_index < len(corrected_waveform):
                corrected_waveform[zero_index:] = np.roll(corrected_waveform[zero_index:], shift=1)
    return corrected_waveform

# Quantum Fourier Transform for phase-space analysis
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Samson Recursive Feedback Formula
def samson_feedback(harmonics, target=0.35, alpha=1.5, iterations=500):
    corrected_harmonics = harmonics.copy()
    for n in range(1, iterations + 1):
        correction = (target - corrected_harmonics[n - 1]) / (alpha * (n + 1))
        feedback = corrected_harmonics[n - 1] * (-0.5) * np.cos(n / np.pi)
        corrected_harmonics[n] = corrected_harmonics[n - 1] + feedback + correction
    return corrected_harmonics

# Generate a simple harmonic state for testing (binary hash in harmonic form)
def generate_harmonics_from_binary(binary_data, expansion_factor=1.5):
    harmonics = np.cumsum(binary_data.astype(np.float64)) * expansion_factor
    return harmonics

# Decode harmonics back to binary (reverse the process)
def decode_harmonics_to_binary(harmonics, expansion_factor=1.5):
    reversed_data = np.diff(harmonics) / expansion_factor
    first_value = harmonics[0] / expansion_factor
    reversed_data = np.insert(reversed_data, 0, first_value) # Include the first corrected value
    return np.round(reversed_data).astype(int)

# Simulate a padded seed stored in H (binary padded for simplicity)
seed_binary = np.array([1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0] + [0] * 112)
seed_harmonics = generate_harmonics_from_binary(seed_binary)
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from mpmath import zetazero

# Generate zeta zeros for the "red wall"
num_zeros = 100
zeta_zeros = [zetazero(n).real for n in range(1, num_zeros + 1)]

# Generate a sine wave to simulate the waveform in H
def generate_sine_wave(length, amplitude=1.0, frequency=1.0, phase=0.0):
    x = np.linspace(0, 2 * np.pi * frequency, length)
    return amplitude * np.sin(x + phase)

# Generate the sine wave
wave_length = 128
sine_wave = generate_sine_wave(wave_length)

# Align the sine wave to the zeta "red wall"
def project_to_zeta_plane(sine_wave, zeta_zeros, length):
    red_wall_projection = np.zeros(length)
    for zero in zeta_zeros:
        index = int((zero / max(zeta_zeros)) * length)
        if 0 <= index < length:
            red_wall_projection[index] = sine_wave[index]
    return red_wall_projection

# Project sine wave onto the zeta "red wall"
projected_wave = project_to_zeta_plane(sine_wave, zeta_zeros, wave_length)
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

Reinitializing the code after reset

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
# Constants for Mark1 framework
```

```
TARGET_HARMONIC = 0.35 # Target alignment value
```

```
QUANTUM_HARMONIC = 0.2 # Quantum focal adjustment angle (in radians)
```

```
MACRO_HARMONIC = 0.33 # Macro focal point (base 10 perception distance)
```

```
MAX_ITERATIONS = 1000 # Maximum iterations for alignment
```

```
TOLERANCE = 1e-4 # Convergence tolerance for harmonic alignment
```

```
# Generate a quantum harmonic spiral
```

```
def quantum_spiral(hash_value, base=2, angle_adjustment=QUANTUM_HARMONIC):
```

```
    """
```

Generate a quantum spiral representation of the hash value with an angular adjustment.

```
    """
```

```
    binary_data = ''.join(format(int(char, 16), f'0{base}b') for char in hash_value)
```

```
    n = len(binary_data)
```

```
    theta = np.linspace(0, 2 * np.pi, n) + angle_adjustment
```

```
    radius = np.linspace(0, 1, n)
```

```
    x = radius * np.cos(theta)
```

```
    y = radius * np.sin(theta)
```

```
    return x, y
```

```
# Recursive harmonic feedback alignment
```

```
def harmonic_feedback_alignment(harmonic_values, target=TARGET_HARMONIC):
```

```
    """
```

Iteratively align harmonics using recursive feedback until convergence to the target.

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

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

```
# Constants for Mark1 framework
TARGET_HARMONIC = 0.35 # Target alignment value
QUANTUM_HARMONIC = 0.2 # Quantum focal adjustment angle (in radians)
TOLERANCE = 1e-6 # Convergence tolerance for harmonic alignment
MAX_ITERATIONS = 50000 # Max iterations for extended looping
```

```
# Generate a quantum harmonic spiral
def quantum_spiral(hash_value, base=2, angle_adjustment=QUANTUM_HARMONIC):
    binary_data = ''.join(format(int(char, 16), f'0{base}b') for char in hash_value)
    n = len(binary_data)
    theta = np.linspace(0, 2 * np.pi, n) + angle_adjustment
    radius = np.linspace(0, 1, n)
    x = radius * np.cos(theta)
    y = radius * np.sin(theta)
    z = radius * np.sin(2 * theta) # Simulating the Z-axis
    return x, y, z, binary_data
```

```
# Recursive harmonic feedback alignment
def harmonic_feedback_alignment(harmonic_values, target=TARGET_HARMONIC):
    aligned_values = harmonic_values.copy()
    history = []
    for _ in range(MAX_ITERATIONS):
        feedback = np.mean(aligned_values)
        history.append(feedback)
        if abs(feedback - target) < TOLERANCE:
            break
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Create a nested ring structure from the binary representation of the hash
def generate_rings_tilted(hash_hex, num_rings=32, tilt_angle=0):
    binary_data = np.array([int(bit) for bit in ".join(format(int(c, 16), '04b') for c in hash_hex)])[::2]
    num_points_per_ring = len(binary_data) // num_rings

    rings = []
    for i in range(num_rings):
        ring_data = binary_data[i * num_points_per_ring: (i + 1) * num_points_per_ring]
        angle = np.linspace(0, 2 * np.pi, num_points_per_ring)
        radius = i / num_rings # Spread rings evenly from center to edge
        x = radius * np.cos(angle)
        y = radius * np.sin(angle)
        z = np.tan(np.radians(tilt_angle)) * radius # Tilt based on angle
        rings.append({"x": x, "y": y, "z": z, "data": ring_data})
    return rings

# Plot the tilted rings
def plot_tilted_rings(rings, title="Tilted View of Nested Rings"):
    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')
    for ring in rings:
        ax.scatter(ring["x"], ring["y"], ring["z"], c=ring["data"], cmap="coolwarm", s=10, alpha=0.8)
    ax.set_title(title, fontsize=14)
    ax.set_xlabel("X-axis")
    ax.set_ylabel("Y-axis")
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
import numpy as np
import hashlib
import matplotlib.pyplot as plt
from scipy.fft import fft
```

```
# Constants for the framework
```

```
EXPANSION_FACTOR = 1.5
QUANTUM_ADJUSTMENT = 0.35
```

```
# Step 1: Generate Quantum Wave from SHA Hash
```

```
def quantum_wave_from_sha(hash_value, base=2):
```

```
    """
```

Generate a quantum waveform from an SHA hash using harmonics.

```
    """
```

```
    binary_data = ''.join(format(int(char, 16), f'0{base}b') for char in hash_value)
    n = len(binary_data)
    theta = np.linspace(0, 2 * np.pi, n)
    radius = np.linspace(0, 1, n)
    x = radius * np.cos(theta)
    y = radius * np.sin(theta)
    z = np.sin(2 * theta) + np.cos(theta / 2) + QUANTUM_ADJUSTMENT
    wave = y + z
    wave_transformed = np.abs(fft(wave))
    return wave_transformed, binary_data
```

```
# Step 2: Apply SHA as a Backflow Modulator (Chaos)
```

```
def sha_backflow(data):
```

```
    """
```

Simulate SHA as a backflow modulator by hashing data.

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

Output image

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Constants for Mark1 framework
EXPANSION_FACTOR = 1.5

# Step 1: Encode the Binary Data into H (Macro to Quantum)
def store_in_H_macro(binary_data, expansion_factor=EXPANSION_FACTOR):
    harmonics = np.cumsum(binary_data.astype(np.float64) * expansion_factor) # Use higher precision
    return harmonics

# Step 2: Encode the Quantum Wave into H (Quantum to Macro)
def store_in_H_quantum(quantum_wave, expansion_factor=EXPANSION_FACTOR):
    harmonics = np.cumsum(quantum_wave * expansion_factor) # Use higher precision for the wave
    return harmonics

# Quantum Wave Generation from Binary Data
def quantum_wave_from_binary(binary_data):
    """
    Generate a quantum wave from binary data using sine and cosine interactions.
    """

    n = len(binary_data)
    theta = np.linspace(0, 2 * np.pi, n)
    radius = np.linspace(0, 1, n)
    wave = radius * (np.sin(theta) + np.cos(theta))
    wave *= binary_data # Modulate the wave with the binary data
    return wave
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
# Define the macro waveform for visualization (assuming it's derived from harmonics or similar data)
macro_waveform = harmonics * 0.8 # Example scaling to create macro waveform (adjust factor as needed)

# Combine quantum and macro data into a unified waveform for visualization
harmonics_combined = harmonics + macro_waveform

# Re-plot with the corrected combination
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

# Parameters for visualization
n_points = len(harmonics_combined)
theta = np.linspace(0, 2 * np.pi, n_points)
radius = np.linspace(0, 1, n_points)

# Extract combined quantum and macro wave for 3D visualization
x_wave = radius * np.cos(theta)
y_wave = radius * np.sin(theta)
z_wave = harmonics_combined # Using combined harmonics

# Plot the 3D representation
ax.plot(x_wave, y_wave, z_wave, label="Combined Waveform", color='blue', lw=2)
ax.scatter(x_wave, y_wave, z_wave, color='red', s=5, label="Nodes")

# Adding labels and legend
ax.set_title("3D Visualization of Combined Quantum and Macro Wave", fontsize=14)
ax.set_xlabel("X-axis", fontsize=12)
ax.set_ylabel("Y-axis", fontsize=12)
ax.set_zlabel("Z-axis (Wave Amplitude)", fontsize=12)
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from scipy.fft import fft

# Constants for Quantum and Harmonic Transformation
EXPANSION_FACTOR = 1.5
QUANTUM_ADJUSTMENT = 0.35

# Function to generate quantum waveform from hash
def quantum_wave_from_hash(hash_value, base=2):
    binary_data = ''.join(format(int(char, 16), f'0{base}b') for char in hash_value)
    n = len(binary_data)
    theta = np.linspace(0, 2 * np.pi, n)
    radius = np.linspace(0, 1, n)
    x = radius * np.cos(theta)
    y = radius * np.sin(theta)
    z = np.sin(2 * theta) + np.cos(theta / 2) # Quantum harmonic combinations
    wave = y + z + QUANTUM_ADJUSTMENT
    return x, y, z, binary_data

# Function to store binary data into H
def store_in_H(binary_data, expansion_factor=EXPANSION_FACTOR):
    harmonics = np.cumsum(binary_data.astype(np.float64) * expansion_factor)
    return harmonics

# Function to superimpose and visualize
def visualize_combined(x, y, z, harmonics, harmonics_x, harmonics_y, harmonics_z):
    fig = plt.figure(figsize=(12, 8))
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Constants for expansion and transformation
EXPANSION_FACTOR = 1.5

# Re-visualize quantum wave and harmonics in one chart
def combined_visualization(quantum_wave, harmonic_data):
    """
    Combines the quantum waveform and harmonic data into a single 3D visualization.

    Parameters:
        quantum_wave (list): A list of values representing the quantum waveform.
        harmonic_data (list): A list of values representing the harmonic data.

    Returns:
        None
    """

    fig = plt.figure(figsize=(14, 10))
    ax = fig.add_subplot(111, projection='3d')

    # Plot the quantum wave
    n = len(quantum_wave)
    x_wave = np.linspace(0, 1, n)
    y_wave = np.sin(quantum_wave / max(quantum_wave))
    z_wave = np.cos(quantum_wave / max(quantum_wave))
    ax.plot(x_wave, y_wave, z_wave, label="Quantum Wave", color='blue')

    # Plot the harmonics
    n_harmonics = len(harmonic_data)
    x_harmonics = np.linspace(0, 1, n_harmonics)
    y_harmonics = np.sin(harmonic_data / max(harmonic_data))
    z_harmonics = np.cos(harmonic_data / max(harmonic_data))
    ax.plot(x_harmonics, y_harmonics, z_harmonics, label="Harmonic Data", color='green')
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from scipy.fft import fft

# Constants for Testing and H Framework
EXPANSION_FACTOR = 1.5

# Step 1: Simulate a Quantum Waveform from the Hash (SHA-like mechanism)
def generate_quantum_wave(hash_value):
    """
    Generate a quantum-like waveform from the input hash value.
    """

    binary_data = ''.join(format(int(char, 16), '04b') for char in hash_value)
    n = len(binary_data)
    theta = np.linspace(0, 2 * np.pi, n)
    radius = np.linspace(0, 1, n)
    x = radius * np.cos(theta)
    y = radius * np.sin(theta)
    z = np.sin(2 * theta) + np.cos(theta / 2)
    wave = y + z
    return x, y, z, wave, binary_data

# Step 2: Store Quantum Data in H
def store_in_H(binary_data, expansion_factor=EXPANSION_FACTOR):
    harmonics = np.cumsum(binary_data.astype(np.float64) * expansion_factor)
    return harmonics

# Step 3: Convert Macro Hash into a Quantum Waveform
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

```
# Constants
```

```
EXPANSION_FACTOR = 1.5
QUANTUM_ADJUSTMENT = 0.35
```

```
# Generate a quantum wave using Samson principles (from hash)
```

```
def quantum_wave_samson(hash_value):
    binary_data = ''.join(format(int(char, 16), '04b') for char in hash_value)
    n = len(binary_data)
    theta = np.linspace(0, 2 * np.pi, n)
    radius = np.linspace(0, 1, n)
    x = radius * np.cos(theta)
    y = radius * np.sin(theta)
    z = np.sin(2 * theta) + np.cos(theta / 2) + QUANTUM_ADJUSTMENT
    return x, y, z
```

```
# Generate a wave from seed + padding binary data
```

```
def macro_wave_from_binary(binary_data):
    n = len(binary_data)
    theta = np.linspace(0, 2 * np.pi, n)
    radius = np.linspace(0, 1, n)
    x = radius * np.cos(theta)
    y = radius * np.sin(theta)
    z = np.sin(2 * theta) + np.cos(theta / 2) + QUANTUM_ADJUSTMENT
    return x, y, z
```

Conversation URL:

<https://chatgpt.com/c/677c9648-b414-8011-8cf7-226300bc3dd3>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

```
# Constants
```

```
EXPANSION_FACTOR = 1.5
QUANTUM_ADJUSTMENT = 0.35
```

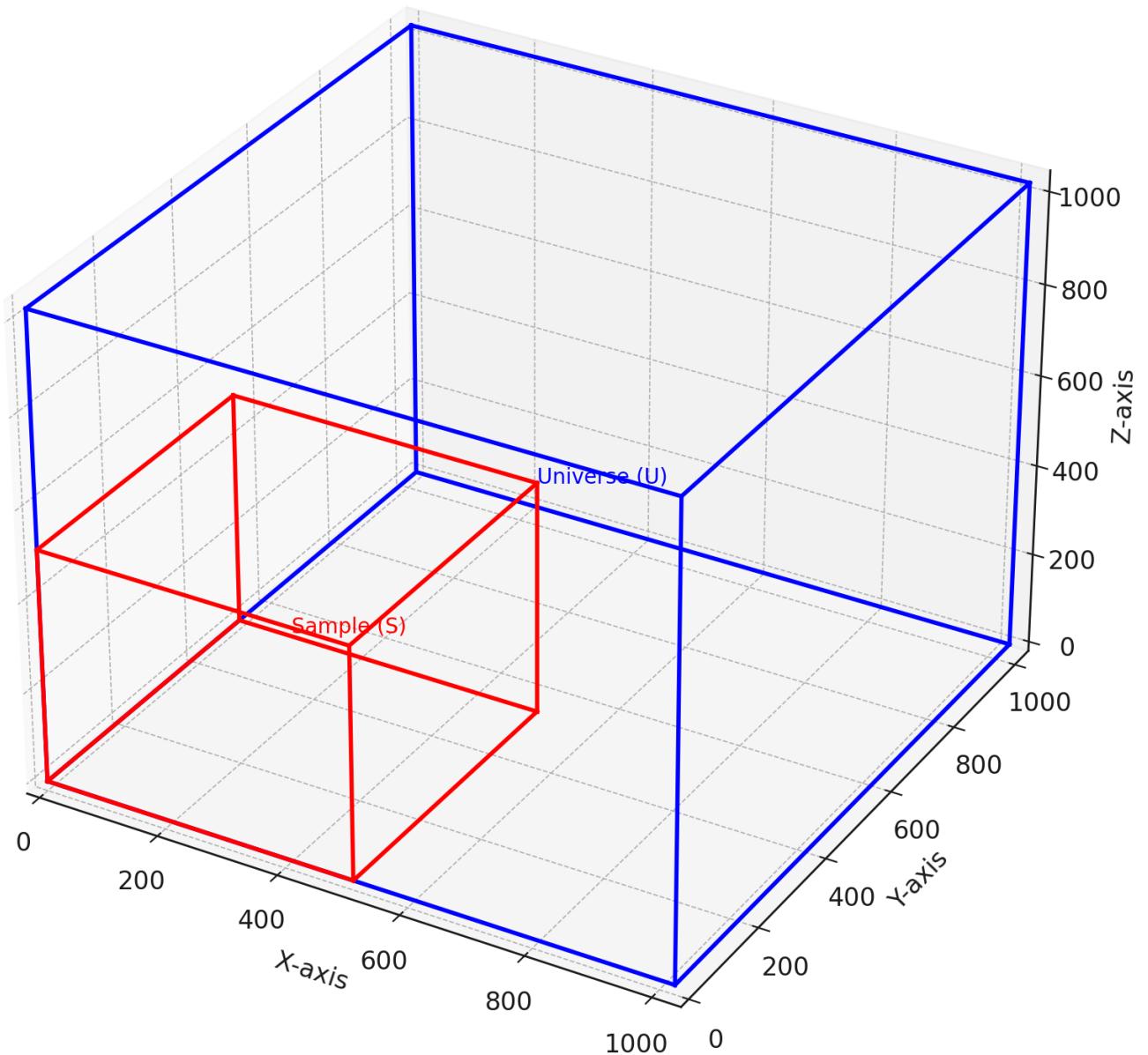
```
# Generate a quantum wave using Samson principles (from hash)
```

```
def quantum_wave_samson(hash_value):
    binary_data = ''.join(format(int(char, 16), '04b') for char in hash_value)
    n = len(binary_data)
    theta = np.linspace(0, 2 * np.pi, n)
    radius = np.linspace(0, 1, n)
    x = radius * np.cos(theta)
    y = radius * np.sin(theta)
    z = np.sin(2 * theta) + np.cos(theta / 2) + QUANTUM_ADJUSTMENT
    return x, y, z
```

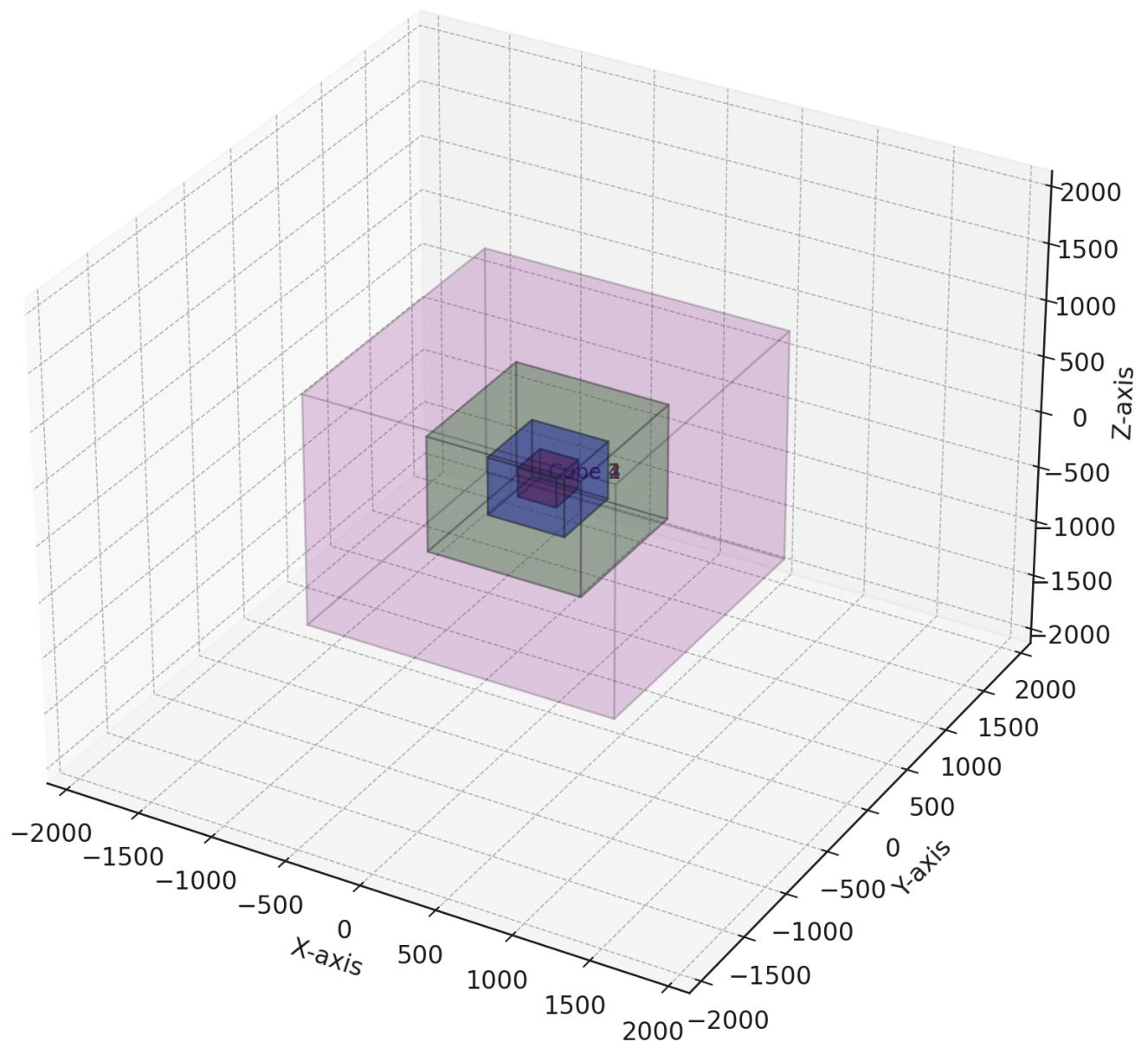
```
# Generate a wave from seed + padding binary data
```

```
def macro_wave_from_binary(binary_data):
    n = len(binary_data)
    theta = np.linspace(0, 2 * np.pi, n)
    radius = np.linspace(0, 1, n)
    x = radius * np.cos(theta)
    y = radius * np.sin(theta)
    z = np.sin(2 * theta) + np.cos(theta / 2) + QUANTUM_ADJUSTMENT
    return x, y, z
```

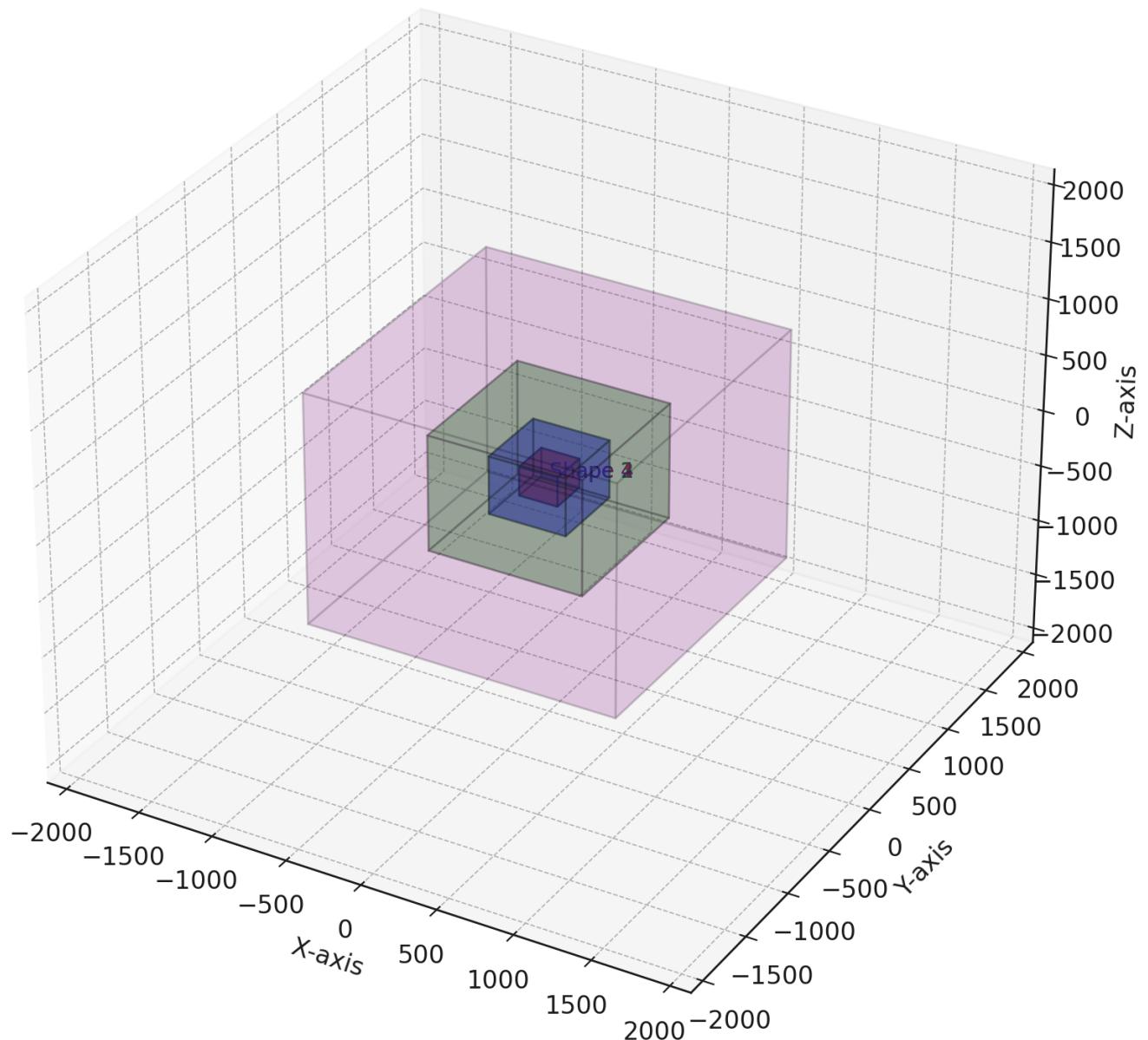
Visualization of Sample (S) and Universe (U) Cubes



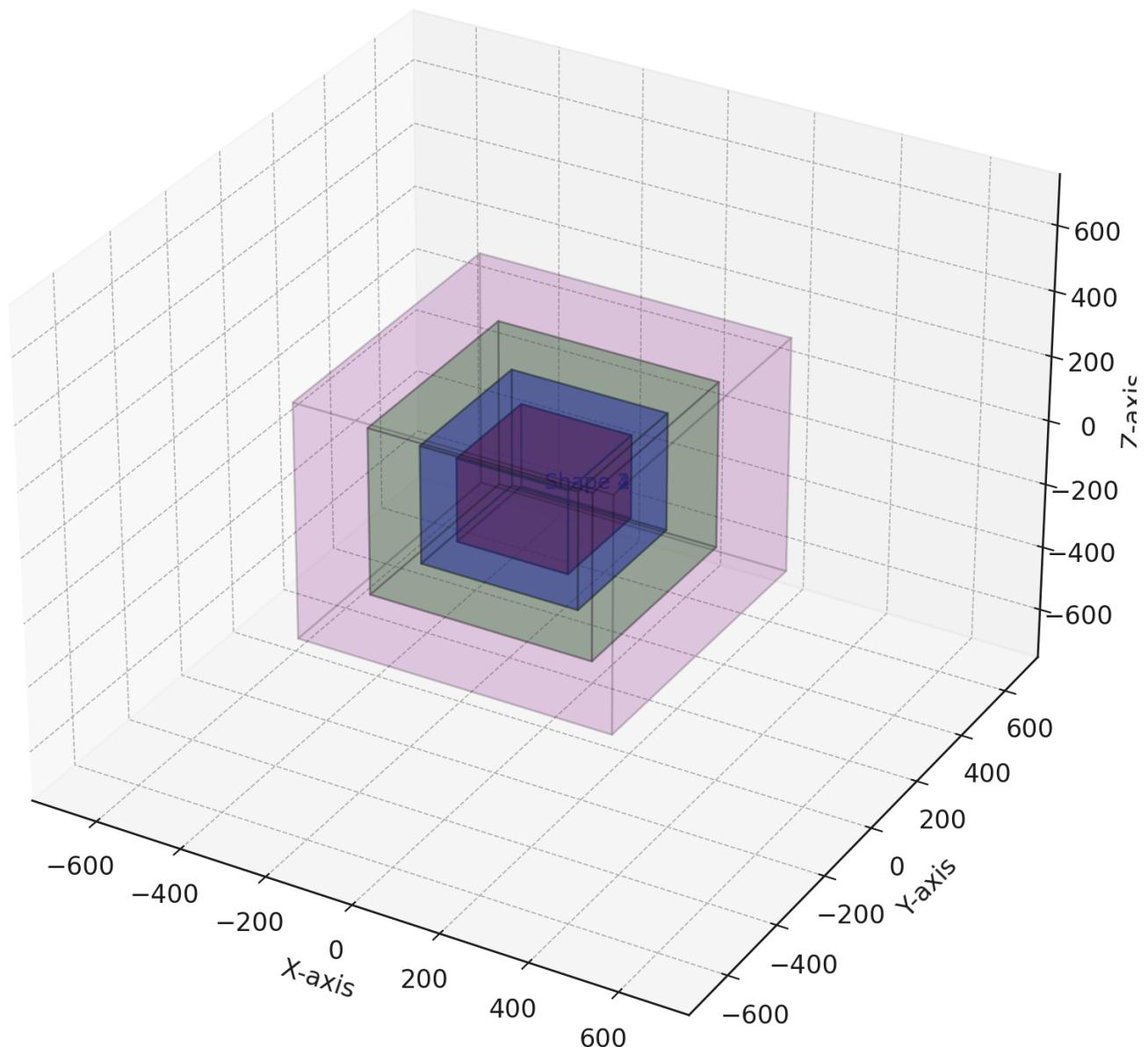
Recursive Nested Cubes (Centered and Corrected)



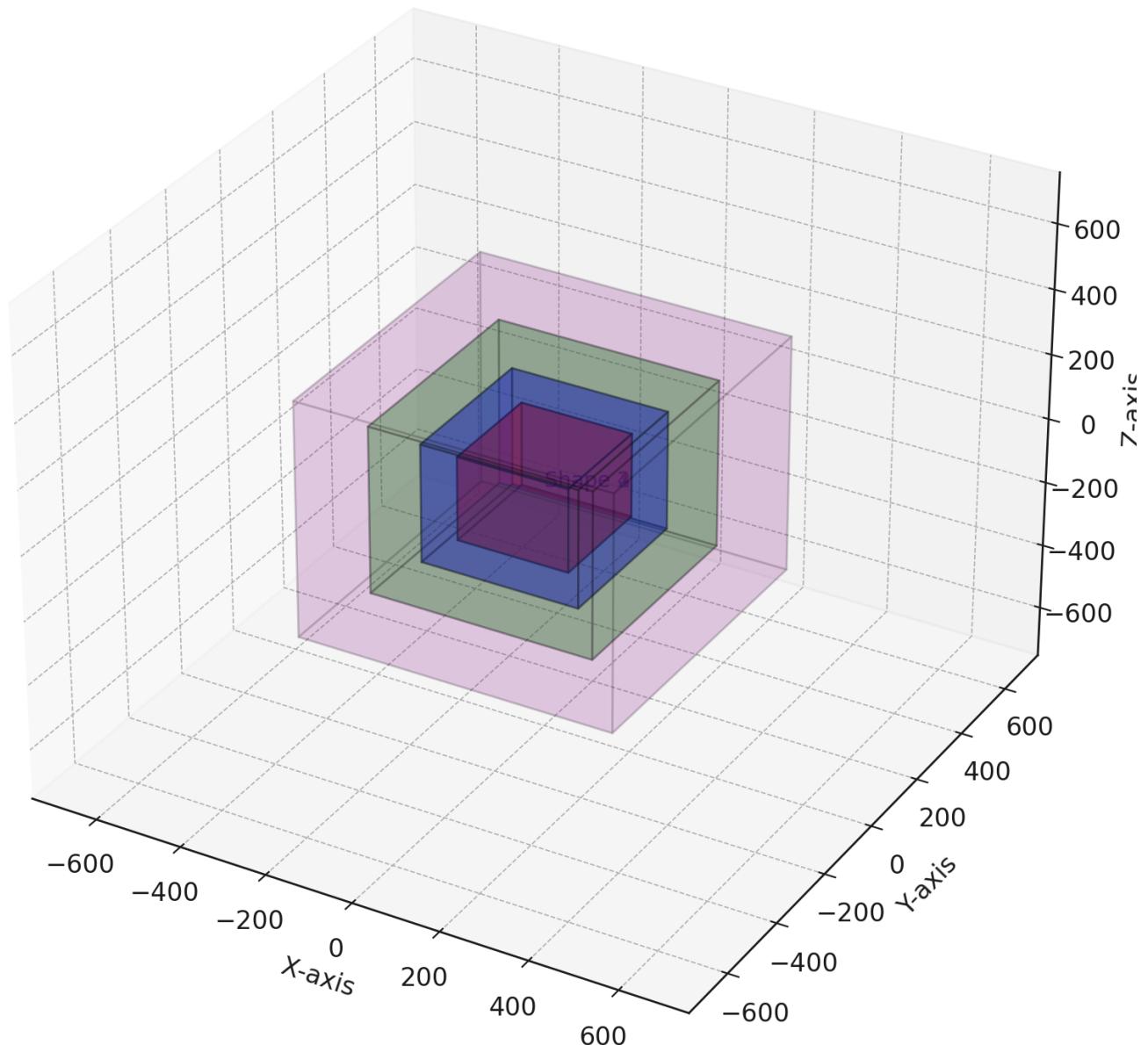
Recursive Nested Shapes (Centered and Corrected)



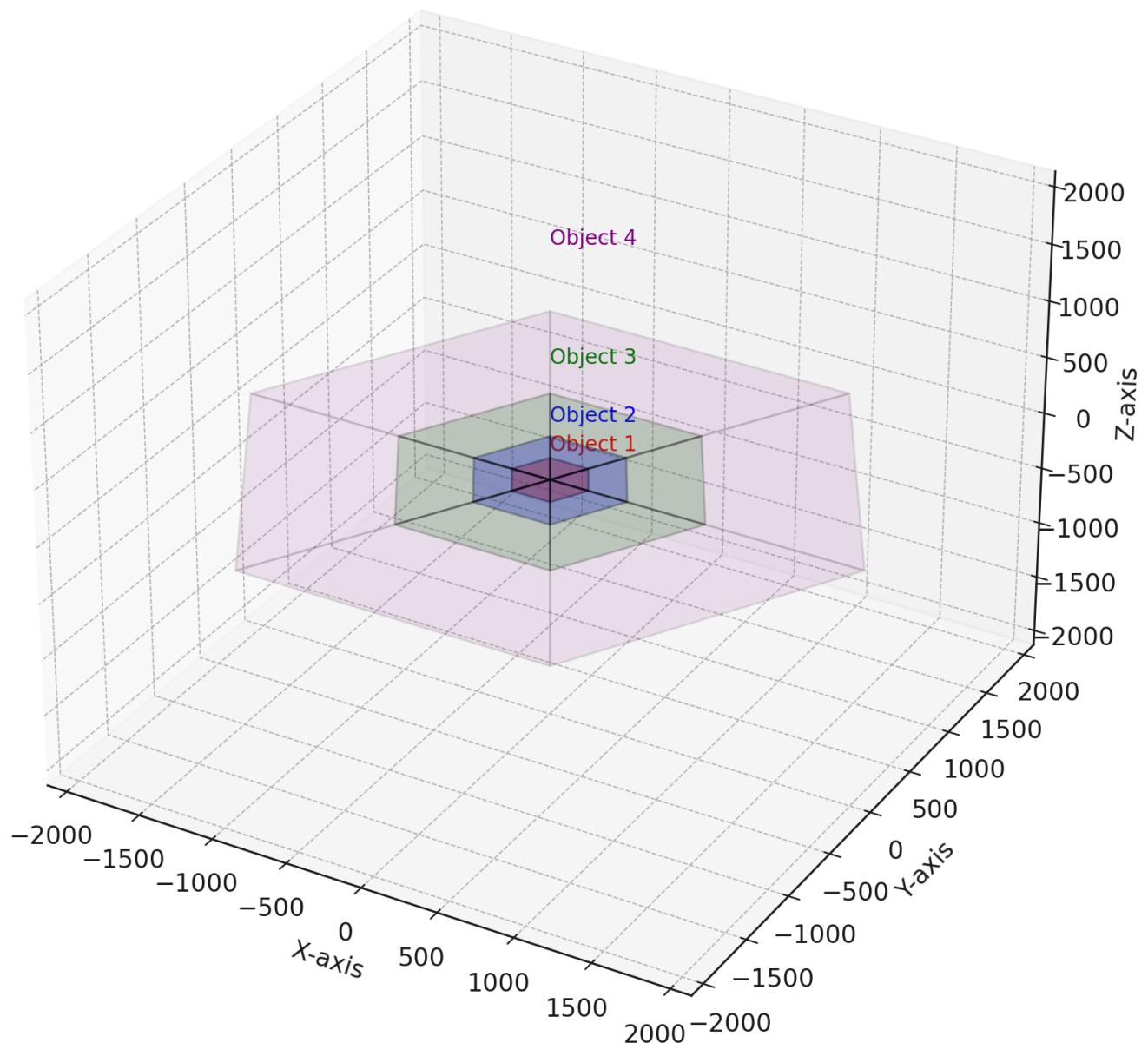
Recursive Reflective Shapes (Nonlinear Growth)



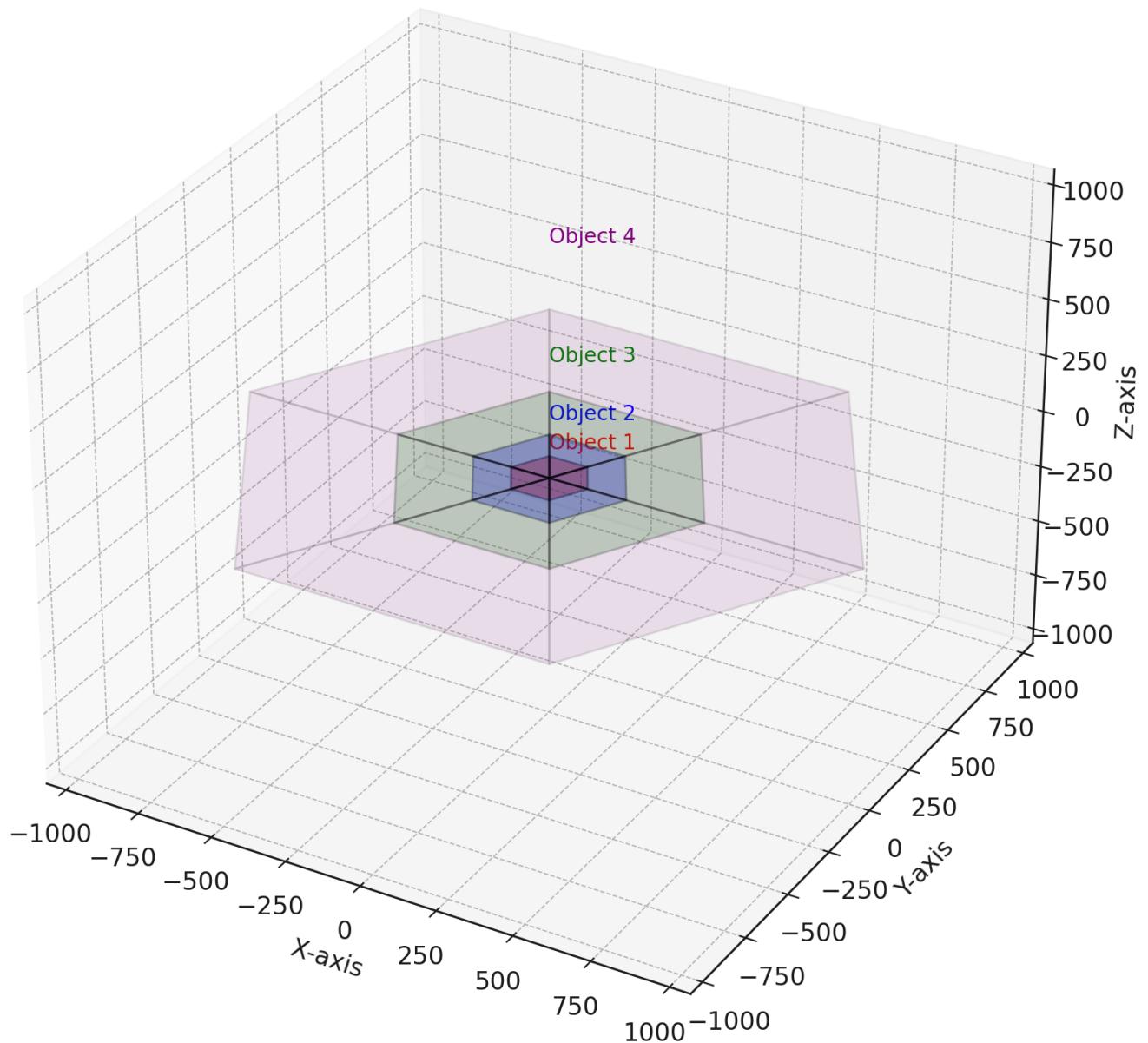
Recursive Generalized Shapes (Dynamic Faces)



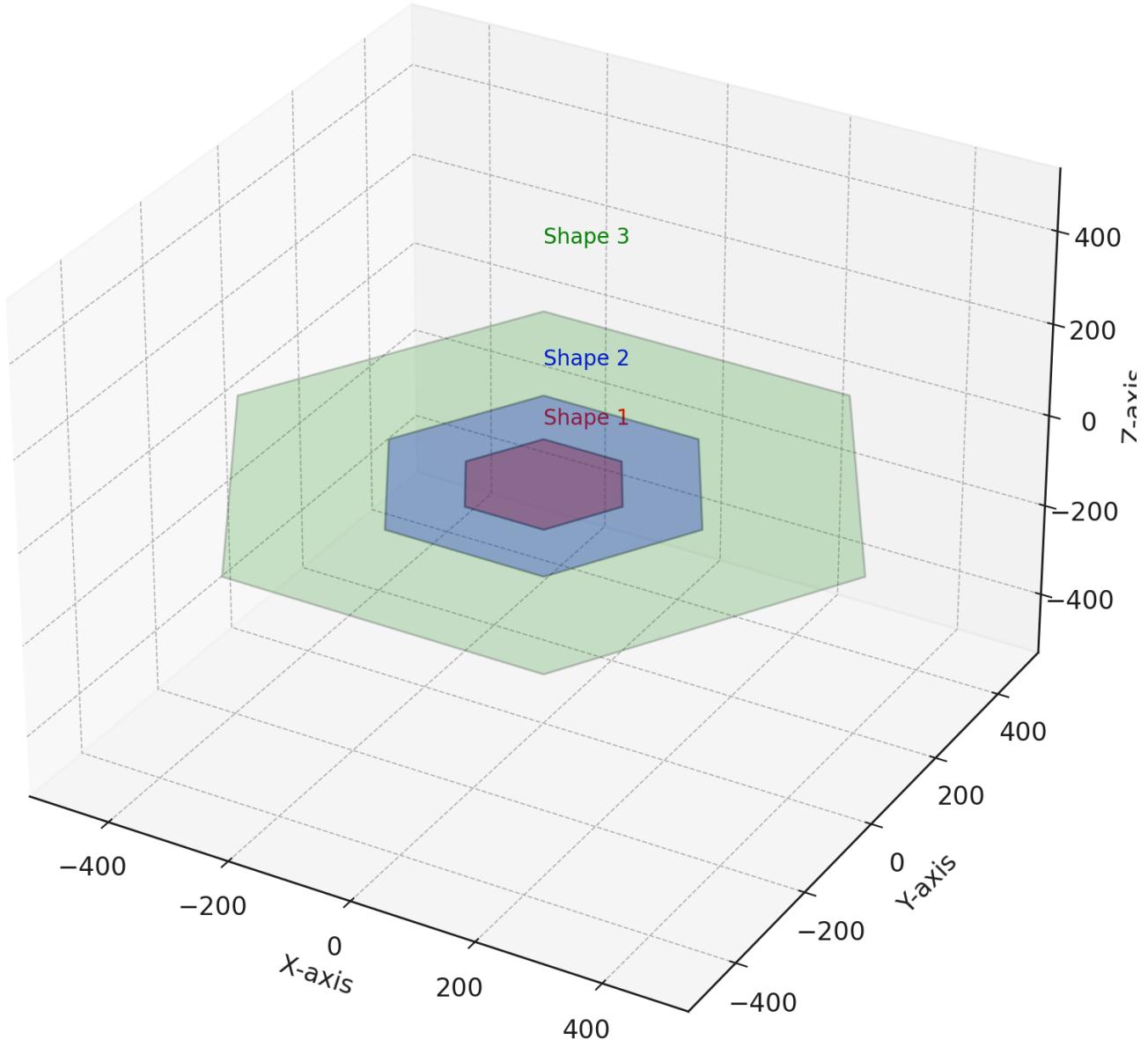
Recursive Waveform Objects with Dynamic Vertices



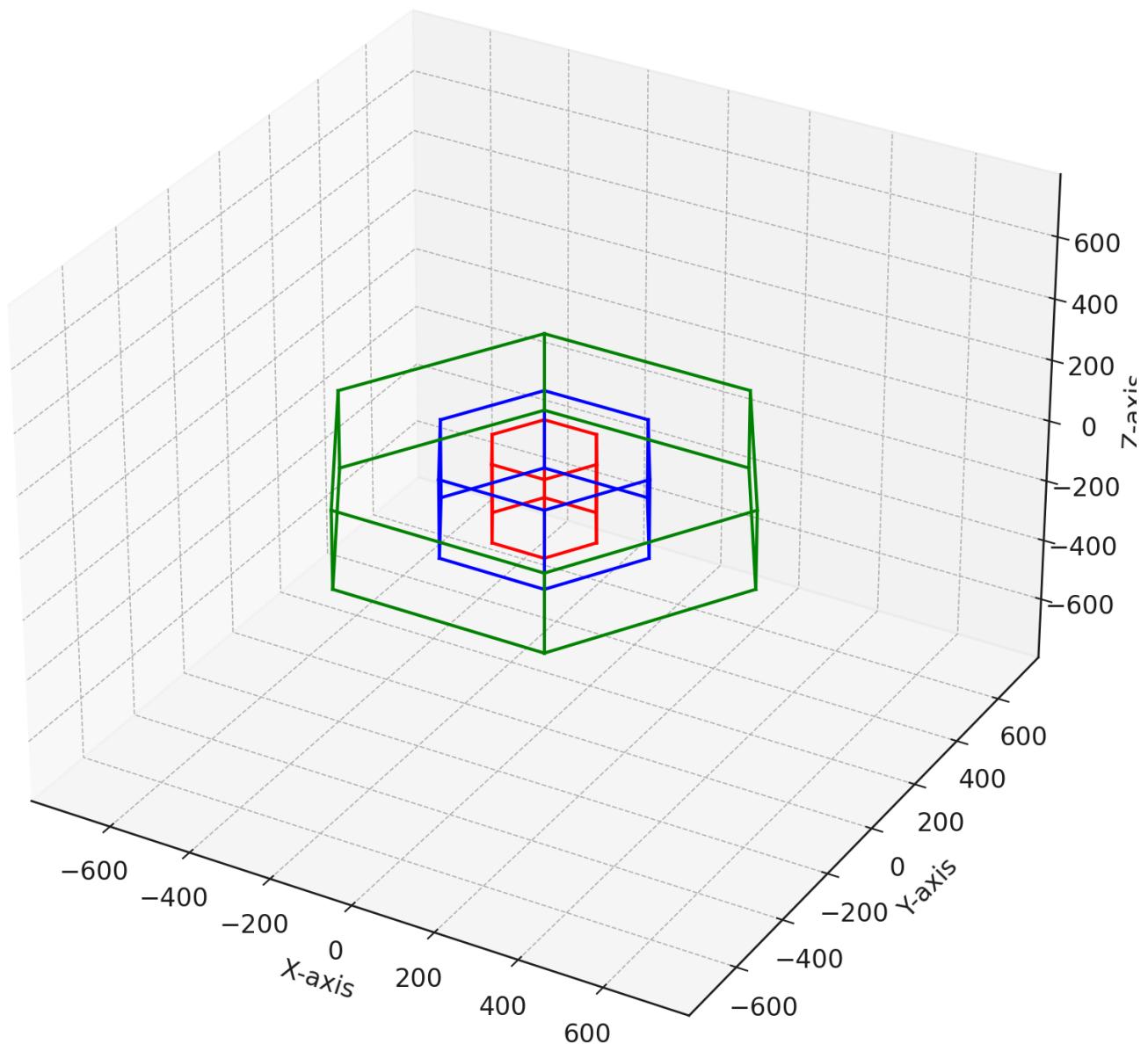
Recursive Objects Grown from (0,0,0)



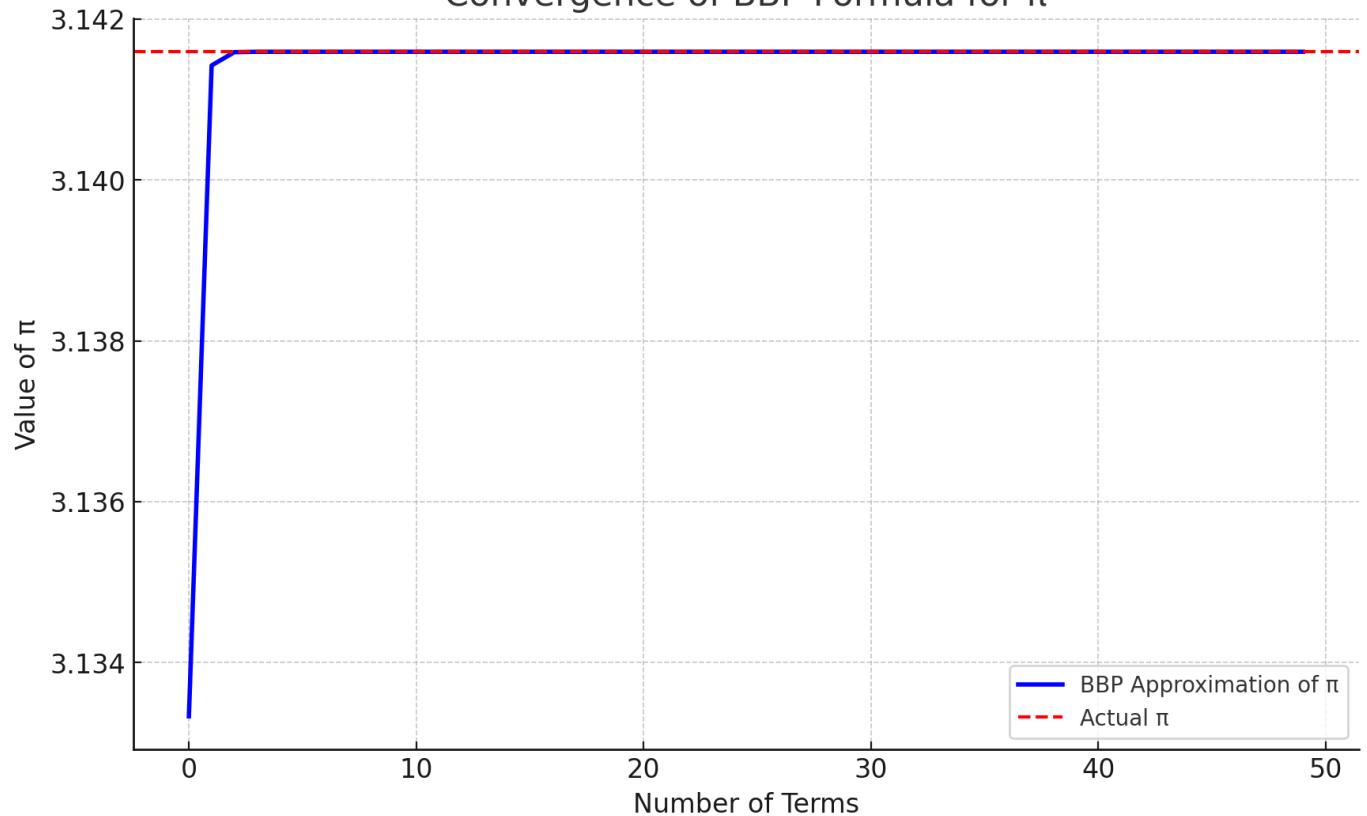
Recursive Shapes Grown from (0,0,0)

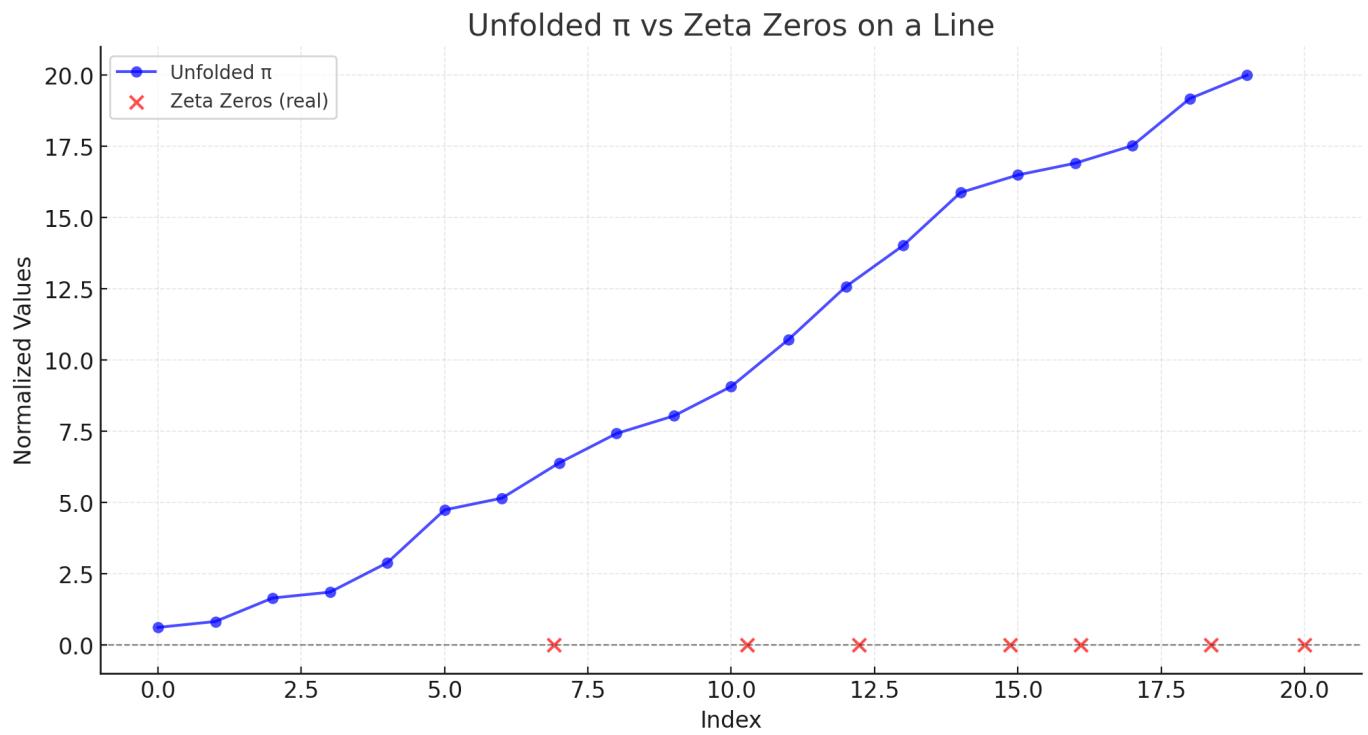


Recursive Shapes Grown from (0,0,0)

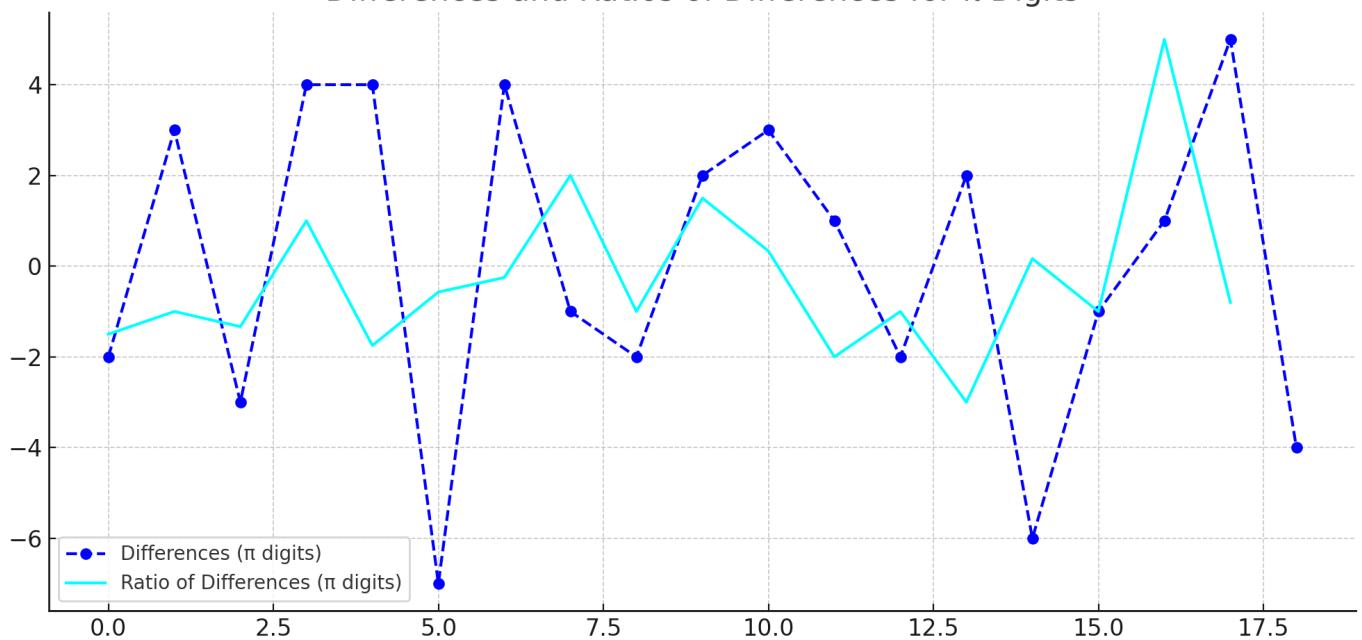


Convergence of BBP Formula for π

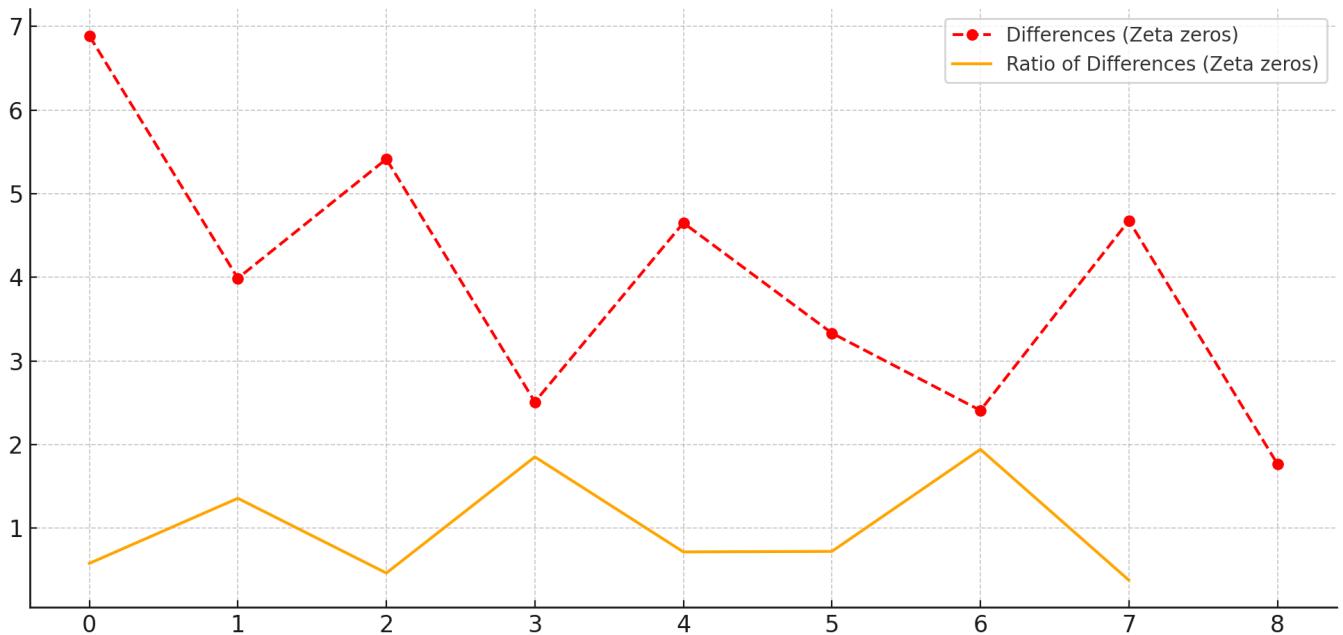


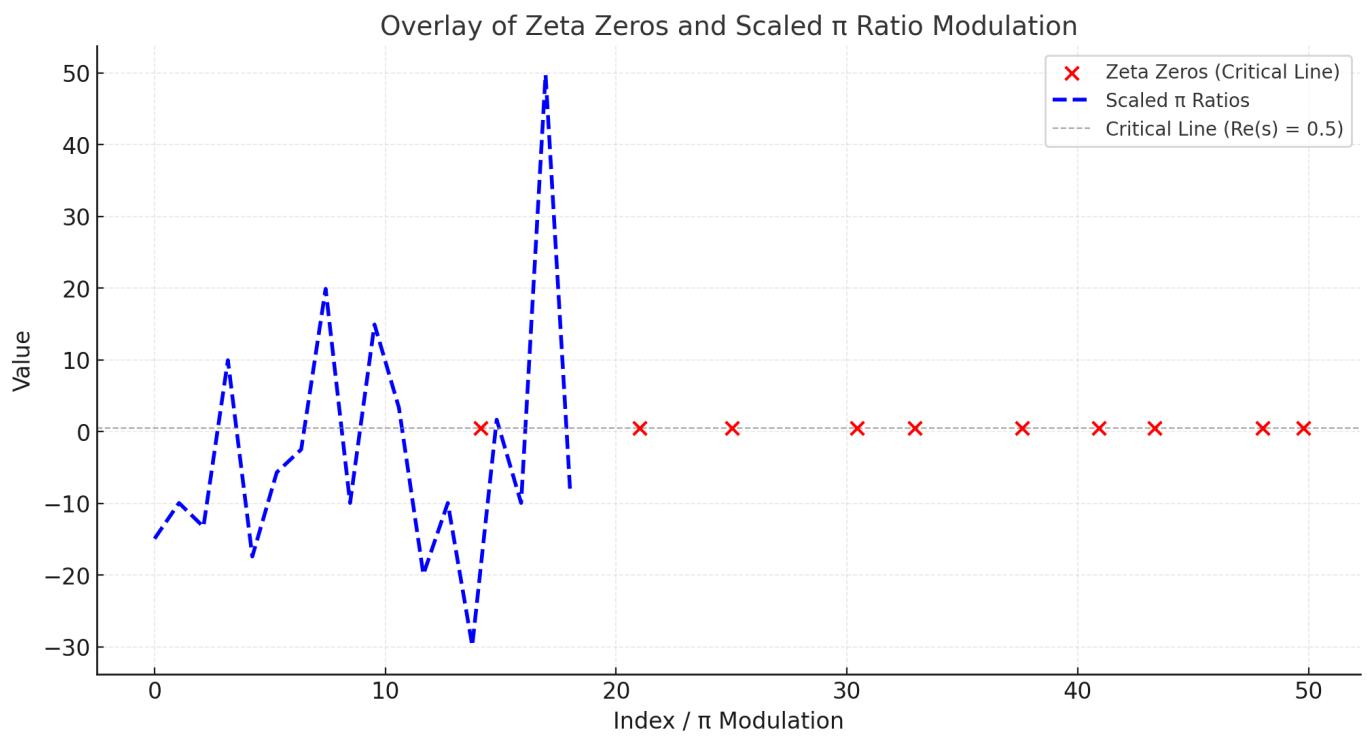


Differences and Ratios of Differences for π Digits

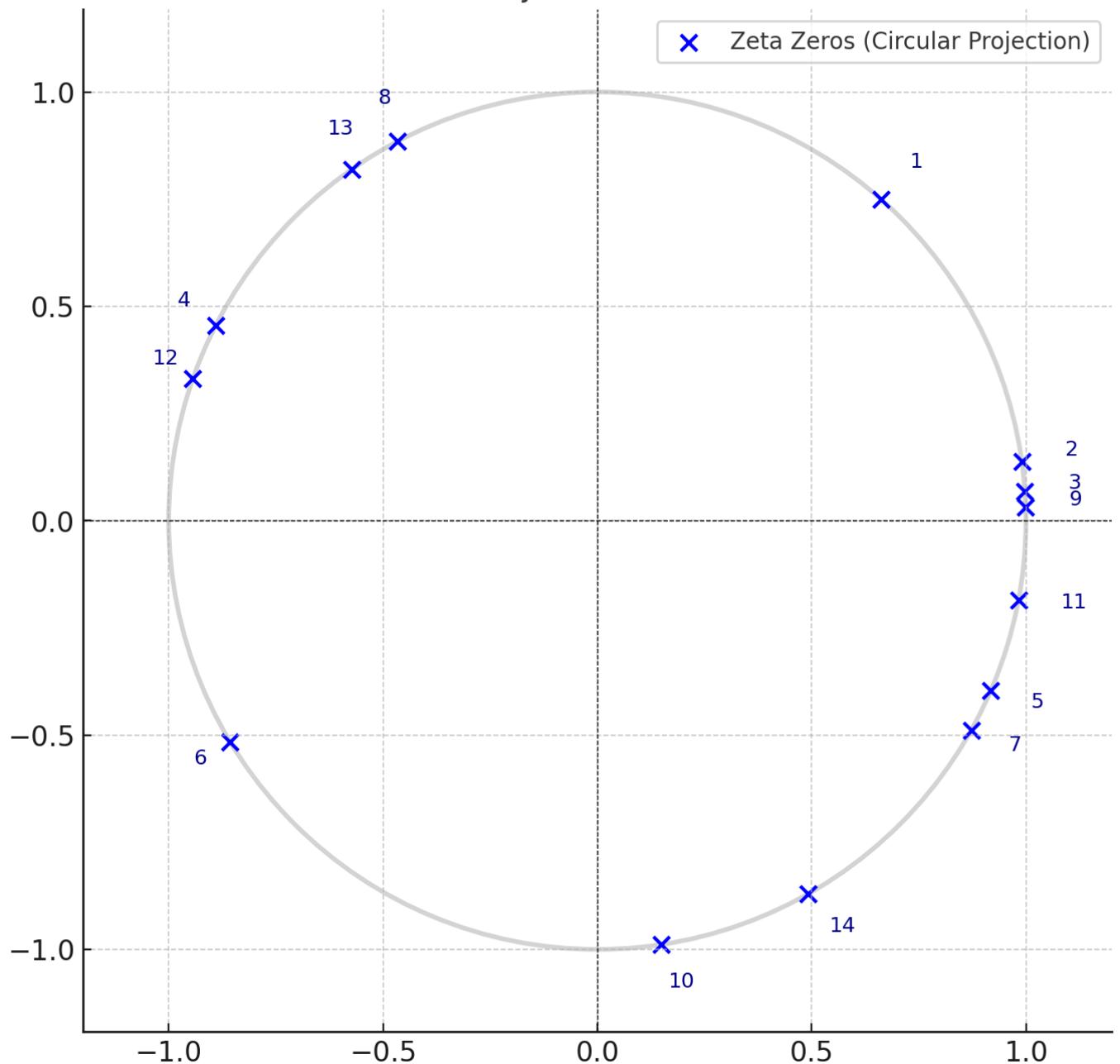


Differences and Ratios of Differences for Zeta Zeros





Circular Projection of Zeta Zeros



Conversation URL:

<https://chatgpt.com/c/6783e51d-5e68-8011-b329-8a8024db41fb>

Title:

Prompt:

```
# Correcting the indexing issue and re-importing necessary libraries
```

```
import matplotlib.pyplot as plt
```

```
from mpl_toolkits.mplot3d import Axes3D
```

```
import numpy as np
```

```
def visualize_cubes(S_dim, U_dim):
```

```
    """
```

Visualizes the Sample and Universe matrices as nested cubes.

Args:

S_dim (tuple): Dimensions of the Sample Matrix (S_x, S_y, S_z).

U_dim (tuple): Dimensions of the Universe Matrix (U_x, U_y, U_z).

```
    """
```

```
# Define the cubes for visualization
```

```
S_x, S_y, S_z = S_dim
```

```
U_x, U_y, U_z = U_dim
```

```
# Define Sample Cube (S) vertices
```

```
S = np.array([
```

```
    [0, 0, 0], [S_x, 0, 0], [S_x, S_y, 0], [0, S_y, 0], # Bottom face
```

```
    [0, 0, S_z], [S_x, 0, S_z], [S_x, S_y, S_z], [0, S_y, S_z] # Top face
```

```
])
```

```
# Define Universe Cube (U) vertices
```

```
U = np.array([
```

```
    [0, 0, 0], [U_x, 0, 0], [U_x, U_y, 0], [0, U_y, 0], # Bottom face
```

```
    [0, 0, U_z], [U_x, 0, U_z], [U_x, U_y, U_z], [0, U_y, U_z] # Top face
```

```
])
```

Conversation URL:

<https://chatgpt.com/c/6783e51d-5e68-8011-b329-8a8024db41fb>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
```

```
def calculate_recursive_cubes(frame_size, recursion_depth=3):
```

"""

Recursively calculates cube dimensions and centers.

Args:

frame_size (int): The size of the initial data cube (smallest).

recursion_depth (int): Number of recursive layers.

Returns:

list: A list of dictionaries containing dimensions and centers for each cube.

"""

```
cubes = []
```

```
# Initialize the innermost cube
```

```
size = frame_size
```

```
center = (0, 0, 0) # Center all cubes at origin for proper alignment
```

```
for _ in range(recursion_depth):
```

Append cube properties

```
cubes.append({"size": size, "center": center})
```

```
# Recursively calculate the next cube
```

```
size *= 2 # Double the size
```

```
return cubes
```

Conversation URL:

<https://chatgpt.com/c/6783e51d-5e68-8011-b329-8a8024db41fb>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
```

```
def calculate_recursive_shapes(seed_size, recursion_depth=3, sides=4):
```

"""

Recursively calculates dimensions and centers for N-sided 3D shapes.

Args:

- seed_size (float): The size of the initial seed shape.
- recursion_depth (int): Number of recursive layers.
- sides (int): Number of sides for the 3D shape (e.g., 4 for cube).

Returns:

- list: A list of dictionaries containing size, center, and other properties for each shape.

"""

```
shapes = []
```

```
# Initialize the seed shape
```

```
size = seed_size
```

```
center = (0, 0, 0) # Growth starts at (0, 0, 0)
```

```
for _ in range(recursion_depth):
```

```
    shapes.append({"size": size, "center": center, "sides": sides})
```

```
# Growth: Double the size, and keep the center at (0, 0, 0)
```

```
    size *= 2
```

```
return shapes
```

Conversation URL:

<https://chatgpt.com/c/6783e51d-5e68-8011-b329-8a8024db41fb>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
```

```
def recursive_growth(seed, recursion_depth, sides=4, step=0):
```

"""

Recursively grows a nested geometric system based on feedback from the previous step.

Args:

- seed (dict): Initial parameters for size, position, and sides.
- recursion_depth (int): How deep to recurse into nested systems.
- sides (int): Number of sides for the shape (default is 4 for a cube).
- step (int): Current recursion step.

Returns:

- list: List of recursively generated shapes.

"""

```
if step >= recursion_depth:
```

```
    return []
```

```
# Calculate the next layer based on reflection from the seed
```

```
current_size = seed["size"]
```

```
next_size = current_size * np.sqrt(2) # Reflective growth, nonlinear scaling
```

```
center = seed["center"]
```

```
# Reflection: Generate the new center based on feedback
```

```
new_center = tuple(c * 1.1 for c in center) # Dynamically shift outward
```

```
# Store the current shape
```

Conversation URL:

<https://chatgpt.com/c/6783e51d-5e68-8011-b329-8a8024db41fb>

Title:

Prompt:

```
def calculate_faces(vertices, sides):
```

"""

Dynamically calculates the faces of a shape based on its vertices.

Args:

vertices (np.array): Vertices of the shape.

sides (int): Number of sides for the shape (e.g., 4 for a cube).

Returns:

list: List of faces, where each face is defined by its vertices.

"""

```
if sides == 4: # Cube logic
```

```
    return [
```

```
        [vertices[i] for i in [0, 1, 2, 3]], # Bottom face
        [vertices[i] for i in [4, 5, 6, 7]], # Top face
        [vertices[i] for i in [0, 1, 5, 4]], # Side face
        [vertices[i] for i in [2, 3, 7, 6]], # Side face
        [vertices[i] for i in [1, 2, 6, 5]], # Side face
        [vertices[i] for i in [0, 3, 7, 4]], # Side face
```

```
    ]
```

```
elif sides == 3: # Tetrahedron logic
```

```
    return [
```

```
        [vertices[i] for i in [0, 1, 2]], # Base face
        [vertices[i] for i in [0, 1, 3]], # Side face
        [vertices[i] for i in [1, 2, 3]], # Side face
        [vertices[i] for i in [0, 2, 3]], # Side face
```

```
    ]
```

else:

```
    raise ValueError(f"Unsupported shape with {sides} sides.")
```

Conversation URL:

<https://chatgpt.com/c/6783e51d-5e68-8011-b329-8a8024db41fb>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
```

```
def generate_polygon_vertices(center, radius, sides):
```

"""

Generate vertices for a regular polygon in 3D space.

Args:

```
center (tuple): The center of the polygon (cx, cy, cz).
radius (float): The radius of the polygon (distance from center to vertex).
sides (int): Number of sides of the polygon.
```

Returns:

```
np.array: Vertices of the polygon in 3D.
```

"""

```
cx, cy, cz = center
vertices = []
for i in range(sides):
    angle = 2 * np.pi * i / sides # Angle for each vertex
    x = cx + radius * np.cos(angle)
    y = cy + radius * np.sin(angle)
    z = cz # Flat on a plane for now (z constant)
    vertices.append([x, y, z])
return np.array(vertices)
```

```
def create_faces(vertices):
```

"""

Conversation URL:

<https://chatgpt.com/c/6783e51d-5e68-8011-b329-8a8024db41fb>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
```

```
def generate_vertices(center, radius, sides, z_func=None):
```

"""

Generate vertices for a regular polygon or 3D shape.

Args:

center (tuple): The center of the shape (cx, cy, cz).

radius (float): Radius of the shape (distance from center to vertex).

sides (int): Number of sides for the polygon.

z_func (function): Function to compute the Z-coordinate for each vertex (optional).

Returns:

np.array: Vertices of the shape in 3D.

"""

```
cx, cy, cz = center
```

```
vertices = []
```

```
for i in range(sides):
```

```
    angle = 2 * np.pi * i / sides # Angle for each vertex
```

```
    x = cx + radius * np.cos(angle)
```

```
    y = cy + radius * np.sin(angle)
```

```
    z = cz if z_func is None else z_func(x, y) # Use z_func if provided
```

```
    vertices.append([x, y, z])
```

```
return np.array(vertices)
```

```
def recursive_growth_objects(center, seed_radius, recursion_depth, sides):
```

Conversation URL:

<https://chatgpt.com/c/6783e51d-5e68-8011-b329-8a8024db41fb>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
```

```
def generate_vertices(center, radius, sides):
```

"""

Generate vertices for a regular n-sided polygon in 3D space.

Args:

center (tuple): Center of the shape (cx, cy, cz).

radius (float): Radius of the shape (distance from center to vertex).

sides (int): Number of sides for the polygon.

Returns:

np.array: Vertices of the polygon in 3D.

"""

```
cx, cy, cz = center
```

```
vertices = []
```

```
for i in range(sides):
```

```
    angle = 2 * np.pi * i / sides # Angle for each vertex
```

```
    x = cx + radius * np.cos(angle)
```

```
    y = cy + radius * np.sin(angle)
```

```
    z = cz # Flat in the XY plane for now
```

```
    vertices.append([x, y, z])
```

```
return np.array(vertices)
```

```
def recursive_shapes(center, seed_radius, recursion_depth, sides):
```

"""

Conversation URL:

<https://chatgpt.com/c/6783e51d-5e68-8011-b329-8a8024db41fb>

Title:

Prompt:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
import numpy as np
```

```
def create_recursive_shapes(frame_size, sides, iterations):
```

Create recursive shapes starting from (0,0,0) based on input frame size, number of sides, and iterations.

```
shapes = []
```

```
# Base radius and height for the polygon based on frame size
```

```
radius = frame_size / 2
```

```
height = frame_size / 2
```

```
for i in range(iterations):
```

```
    current_radius = radius * (2**i)
```

```
    vertices = []
```

```
    for j in range(sides):
```

```
        angle = (2 * np.pi / sides) * j
```

```
        x = current_radius * np.cos(angle)
```

```
        y = current_radius * np.sin(angle)
```

```
        z = 0
```

```
        vertices.append([x, y, z])
```

```
# Adding top and bottom vertices for 3D polyhedron representation
```

```
top_vertices = [[x, y, height] for x, y, _ in vertices]
```

```
bottom_vertices = [[x, y, -height] for x, y, _ in vertices]
```

```
shapes.append((top_vertices, bottom_vertices))
```

Conversation URL:

<https://chatgpt.com/c/6783e51d-5e68-8011-b329-8a8024db41fb>

Title:

Prompt:

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

```
def compute_bbp_pi(n_terms):
```

Compute the approximation of pi using the Bailey–Borwein–Plouffe (BBP) formula.

Args:

n_terms (int): Number of terms in the BBP series.

Returns:

float: Approximation of pi.

list: Partial sums of pi for each term.

```
pi_approx = 0  
partial_sums = []  
for k in range(n_terms):  
    term = (1 / (16**k)) * (  
        4 / (8*k + 1) - 2 / (8*k + 4) - 1 / (8*k + 5) - 1 / (8*k + 6)  
    )  
    pi_approx += term  
    partial_sums.append(pi_approx)  
return pi_approx, partial_sums
```

```
def visualize_bbp_convergence(n_terms):
```

Visualize the convergence of the BBP formula for pi.

Args:

Conversation URL:

<https://chatgpt.com/c/6783e51d-5e68-8011-b329-8a8024db41fb>

Title:

Prompt:

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

# Load Zeta zeros (real parts only for simplicity) and simulate some Pi digits (expand later as needed)
# Assuming real Zeta zeros
zeta_zeros = np.array([14.134725, 21.022040, 25.010858, 30.424876, 32.935062, 37.586178, 40.918719])
# Simulated π digits, representing first 20 steps for now
pi_digits = np.array([3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3, 2, 3, 8, 4])

# Extend Pi digits to create "unfolding" structure - linear progression of decimal values
pi_unfolded = np.cumsum(pi_digits)

# Normalize Zeta zeros and π values to align scales (optional, for visualization purposes)
zeta_normalized = zeta_zeros / np.max(zeta_zeros) * len(pi_unfolded)
pi_normalized = pi_unfolded / np.max(pi_unfolded) * len(pi_unfolded)

# Visualization
plt.figure(figsize=(12, 6))
plt.plot(pi_normalized, label="Unfolded π", linestyle='-', color='blue', marker='o', alpha=0.7)
plt.scatter(zeta_normalized, np.zeros_like(zeta_normalized), label="Zeta Zeros (real)", color='red', s=50, alpha=0.7)
plt.axhline(0, color='black', linestyle='--', linewidth=0.8, alpha=0.5)
plt.xlabel("Index")
plt.ylabel("Normalized Values")
plt.title("Unfolded π vs Zeta Zeros on a Line")
plt.legend()
plt.grid(alpha=0.3)
plt.show()
```

Conversation URL:

<https://chatgpt.com/c/6783e51d-5e68-8011-b329-8a8024db41fb>

Title:

Prompt:

```
# Reinitialize data
```

```
pi_digits = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3, 2, 3, 8, 4]
```

```
zeta_real_parts = [14.134725, 21.02204, 25.010857, 30.424876, 32.935062,
```

```
    37.586178, 40.918719, 43.327073, 48.00515, 49.773832]
```

```
# Recalculate differences
```

```
pi_differences = np.diff(pi_digits)
```

```
zeta_differences = np.diff(zeta_real_parts)
```

```
# Recalculate ratios with explicit casting to float
```

```
pi_ratio_of_differences = np.divide(
```

```
    pi_differences[1:].astype(float),
```

```
    pi_differences[:-1].astype(float),
```

```
    out=np.zeros_like(pi_differences[1:], dtype=float),
```

```
    where=pi_differences[:-1] != 0
```

```
)
```

```
zeta_ratio_of_differences = np.divide(
```

```
    zeta_differences[1:].astype(float),
```

```
    zeta_differences[:-1].astype(float),
```

```
    out=np.zeros_like(zeta_differences[1:], dtype=float),
```

```
    where=zeta_differences[:-1] != 0
```

```
)
```

```
# Plotting the results
```

```
fig, axs = plt.subplots(2, 1, figsize=(10, 10))
```

```
# Plot the differences and ratios for π digits
```

```
axs[0].plot(pi_differences, label="Differences (π digits)", marker="o", linestyle="--", color="blue")
```

Conversation URL:

<https://chatgpt.com/c/6783e51d-5e68-8011-b329-8a8024db41fb>

Title:

Prompt:

```
# Known Zeta zeros (imaginary parts on the critical line)
zeta_zeros_imaginary = [14.134725, 21.02204, 25.010857, 30.424876, 32.935062,
                        37.586178, 40.918719, 43.327073, 48.00515, 49.773832]

# π digits and ratios of changes
pi_digits = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3, 2, 3, 8, 4]
pi_differences = np.diff(pi_digits) # Consecutive differences in π digits
pi_ratio_of_differences = np.divide(
    pi_differences[1:].astype(float),
    pi_differences[:-1].astype(float),
    out=np.zeros_like(pi_differences[1:], dtype=float),
    where=pi_differences[:-1] != 0
)

# Scale the π ratios to the Zeta zeros' range for better overlay visualization
scaled_pi_ratios = pi_ratio_of_differences / np.max(pi_ratio_of_differences) * max(zeta_zeros_imaginary)

# Generate x-coordinates for π-driven curve
x_pi = np.linspace(0, len(scaled_pi_ratios), len(scaled_pi_ratios))

# Visualization
plt.figure(figsize=(12, 6))

# Plot Zeta zeros on the critical line
plt.scatter(zeta_zeros_imaginary, [0.5] * len(zeta_zeros_imaginary),
            color='red', label='Zeta Zeros (Critical Line)', zorder=5, s=50)

# Overlay scaled π ratio curve
plt.plot(x_pi, scaled_pi_ratios, label='Scaled π Ratios', color='blue', linestyle='--', linewidth=2)
```

Conversation URL:

<https://chatgpt.com/c/6783e51d-5e68-8011-b329-8a8024db41fb>

Title:

Prompt:

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

# Generate the first few non-trivial Zeta zeros (approximated values)
# These values are commonly known zeros along the critical line Re(s) = 1/2
zeta_zeros = [
    14.134725, 21.022040, 25.010858, 30.424876, 32.935061, 37.586178, 40.918719,
    43.327073, 48.005150, 49.773832, 52.970321, 56.446247, 59.347044, 60.831778
]

# Convert Zeta zeros into circular projection angles
# Scale using 2 * pi and modulo to fit into a circular framework
angles = [(2 * np.pi * zero) % (2 * np.pi) for zero in zeta_zeros]

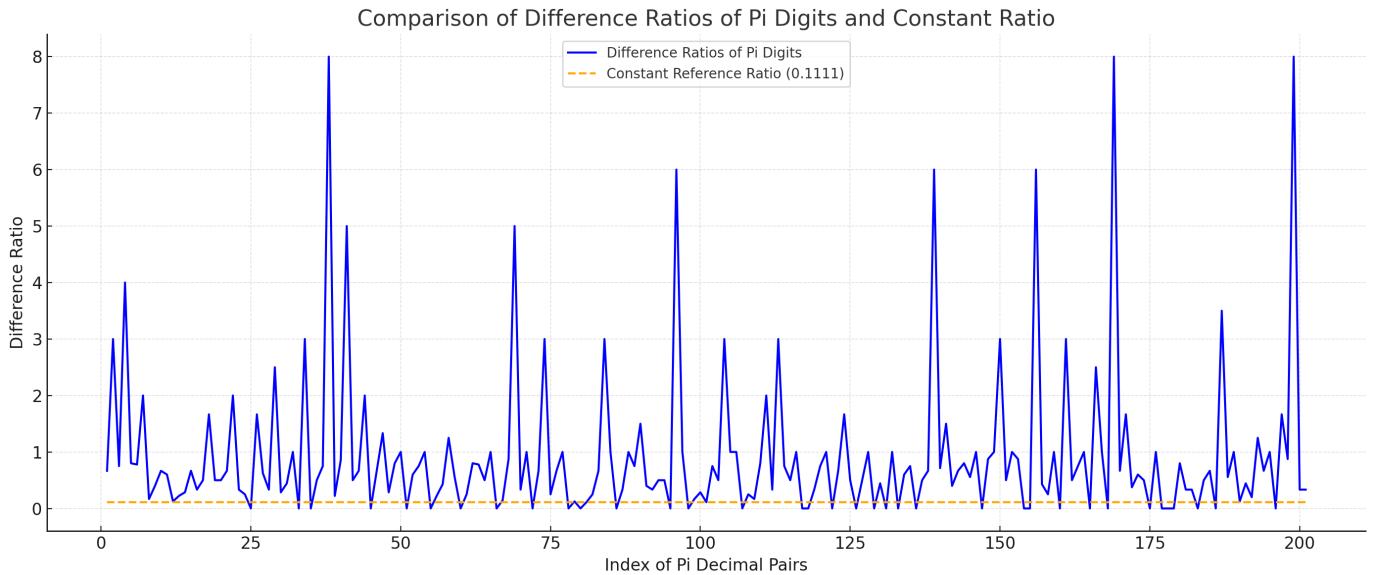
# Generate circular coordinates for Zeta zeros
x_coords = [np.cos(angle) for angle in angles]
y_coords = [np.sin(angle) for angle in angles]

# Create the circular projection plot
fig, ax = plt.subplots(figsize=(8, 8))

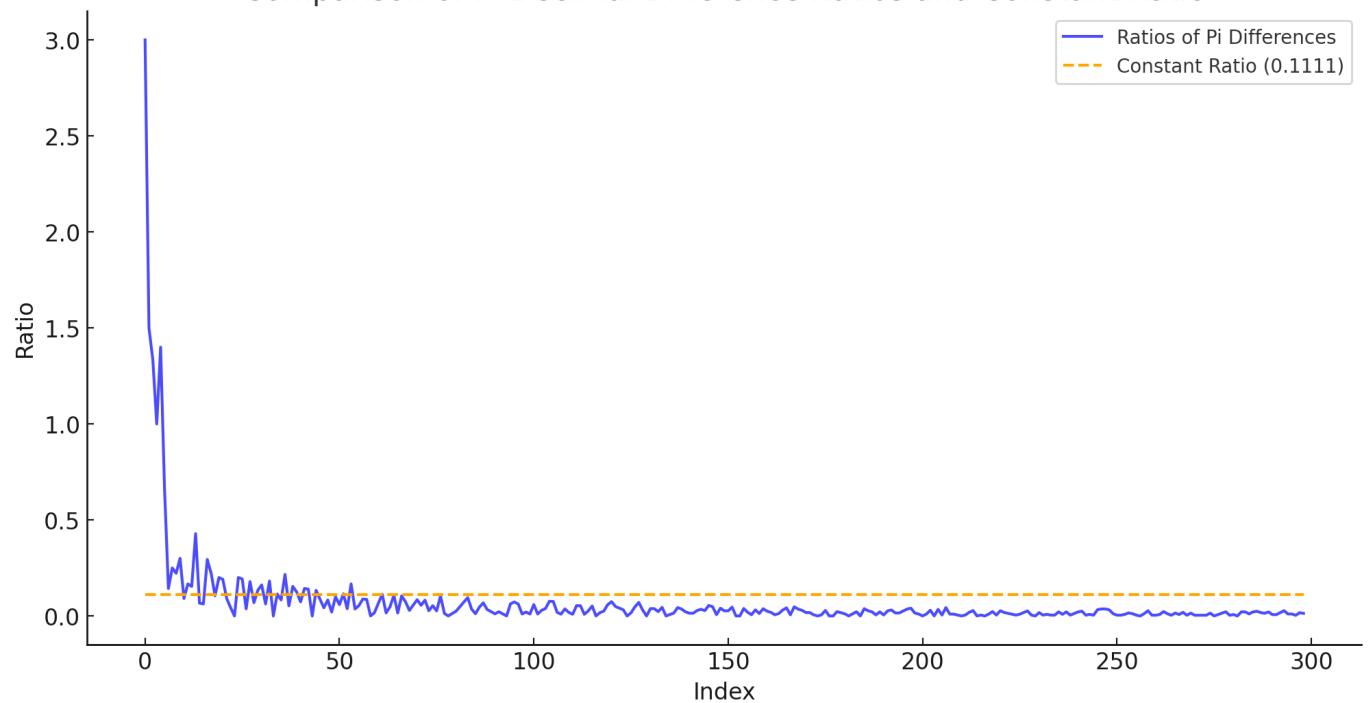
# Draw the unit circle
circle = plt.Circle((0, 0), 1, color='lightgray', fill=False, linewidth=2)
ax.add_artist(circle)

# Plot Zeta zero projections
ax.scatter(x_coords, y_coords, color='blue', label='Zeta Zeros (Circular Projection)', s=50)

# Annotate points
```



Comparison of Pi Decimal Difference Ratios and Constant Ratio



Conversation URL:

<https://chatgpt.com/c/6785ef81-53cc-8011-9f57-170dada5a01d>

Title:

Comparison of Difference Ratios of Pi Digits and Constant Ratio

Prompt:

```
import matplotlib.pyplot as plt
```

```
import numpy as np
```

```
# Function to calculate difference ratios of Pi's digits
```

```
def calculate_difference_ratios(digits):
```

```
    return [abs(digits[i] - digits[i-1]) / digits[i-1] if digits[i-1] != 0 else 0 for i in range(1, len(digits))]
```

```
# Generate Pi's first 300 digits as integers
```

```
pi_digits = [int(d) for d in "31415926535897932384626433832795028841971693993751058209749445923078164062862089"]
```

```
pi_digits += [int(d) for d in "8214808651328230664709384460955058223172535940812848111745028410270193852110555"]
```

```
# Calculate difference ratios
```

```
difference_ratios = calculate_difference_ratios(pi_digits[:300])
```

```
# Create the constant reference ratio
```

```
reference_ratio = [0.1111] * len(difference_ratios)
```

```
# Plot
```

```
plt.figure(figsize=(14, 6))
```

```
plt.plot(range(1, len(difference_ratios) + 1), difference_ratios, label="Difference Ratios of Pi Digits", color="blue")
```

```
plt.plot(range(1, len(difference_ratios) + 1), reference_ratio, linestyle='--', label="Constant Reference Ratio (0.1111)", color="orange")
```

```
# Highlight features
```

```
plt.xlabel("Index of Pi Decimal Pairs")
```

```
plt.ylabel("Difference Ratio")
```

```
plt.title("Comparison of Difference Ratios of Pi Digits and Constant Ratio")
```

```
plt.legend()
```

```
plt.grid(alpha=0.4)
```

```
plt.tight_layout()
```

Conversation URL:

<https://chatgpt.com/c/6785ef81-53cc-8011-9f57-170dada5a01d>

Title:

Comparison of Pi Decimal Difference Ratios and Constant Ratio

Prompt:

```
import matplotlib.pyplot as plt
```

```
import numpy as np
```

```
# Calculate the difference ratios for the first 300 decimal digits of π
```

```
from mpmath import mp
```

```
mp.dps = 310 # Set precision
```

```
pi_digits = [int(d) for d in str(mp.pi)[2:302]] # Extract first 300 decimal digits of π
```

```
# Calculate differences
```

```
differences = [pi_digits[i] - pi_digits[i - 1] for i in range(1, len(pi_digits))]
```

```
ratios = [abs(differences[i] / (i + 1)) for i in range(len(differences))]
```

```
# Generate the constant ratio line
```

```
constant_ratio = [0.1111 for _ in range(len(ratios))]
```

```
# Plotting the results
```

```
plt.figure(figsize=(12, 6))
```

```
plt.plot(ratios, label="Ratios of Pi Differences", color="blue", alpha=0.7)
```

```
plt.plot(constant_ratio, label="Constant Ratio (0.1111)", color="orange", linestyle="--")
```

```
plt.title("Comparison of Pi Decimal Difference Ratios and Constant Ratio")
```

```
plt.xlabel("Index")
```

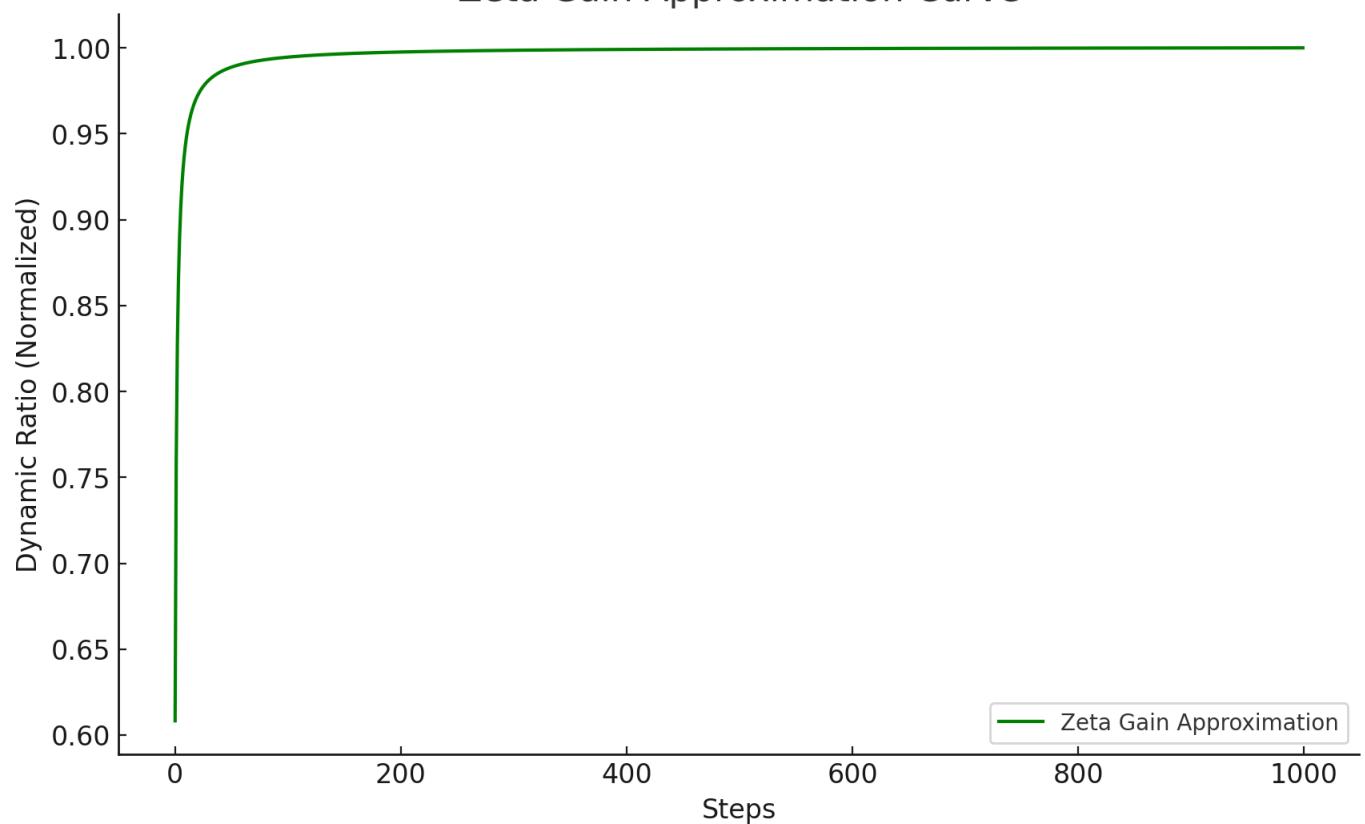
```
plt.ylabel("Ratio")
```

```
plt.legend()
```

```
plt.grid()
```

```
plt.show()
```

Zeta Gain Approximation Curve



Conversation URL:

<https://chatgpt.com/c/6786db90-e554-8011-9f1e-f4d78dde7a8e>

Title:

Prompt:

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

# Define Zeta function approximation for the recursive relationship
def zeta_gain_approximation(n, s=2):
    return np.cumsum(1 / np.power(np.arange(1, n+1), s))

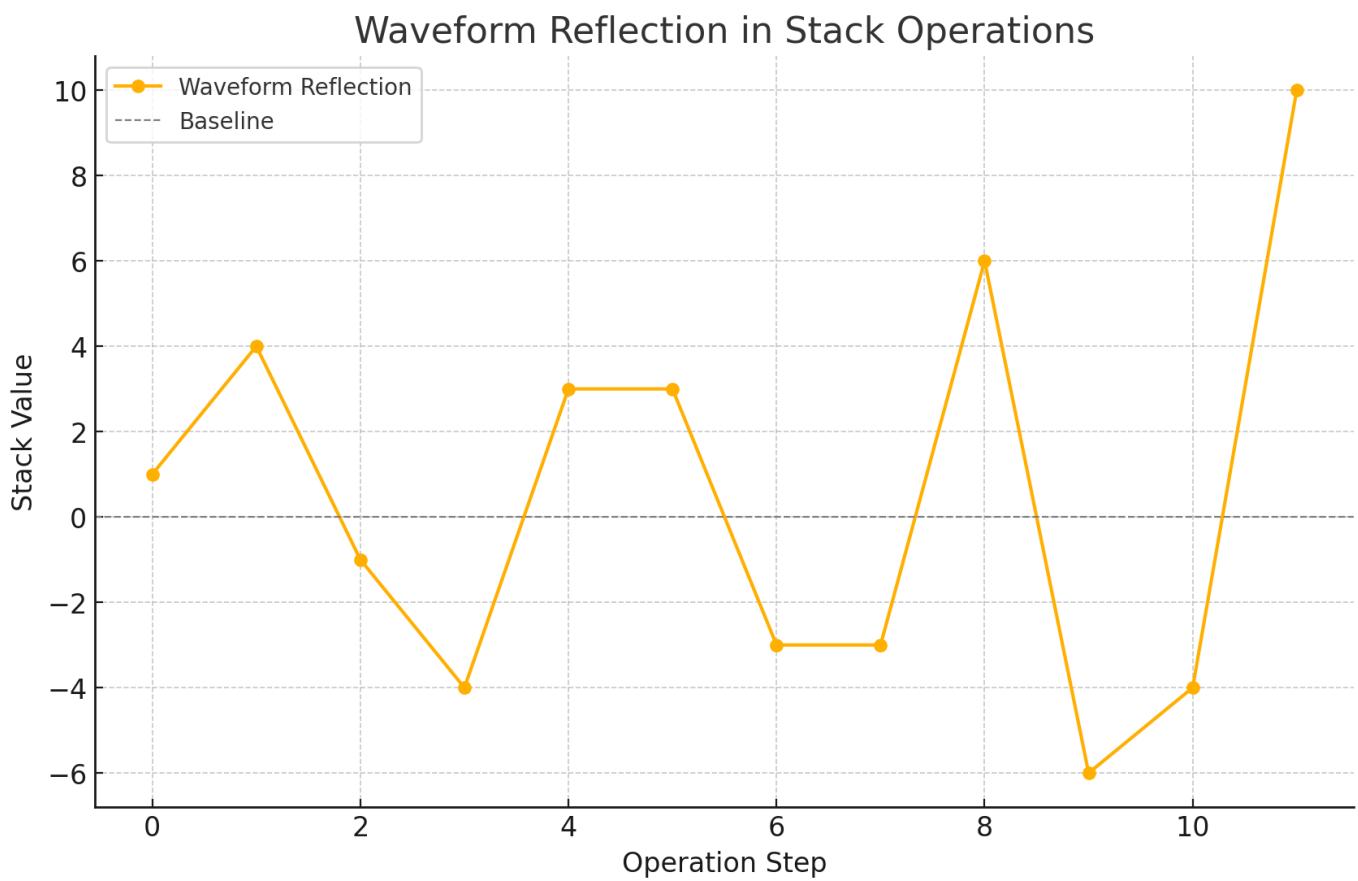
# Generate Zeta values
n_values = 1000
zeta_values = zeta_gain_approximation(n_values)

# Normalize Zeta values for scaling (to compare with base gains)
zeta_normalized = zeta_values / zeta_values[-1]

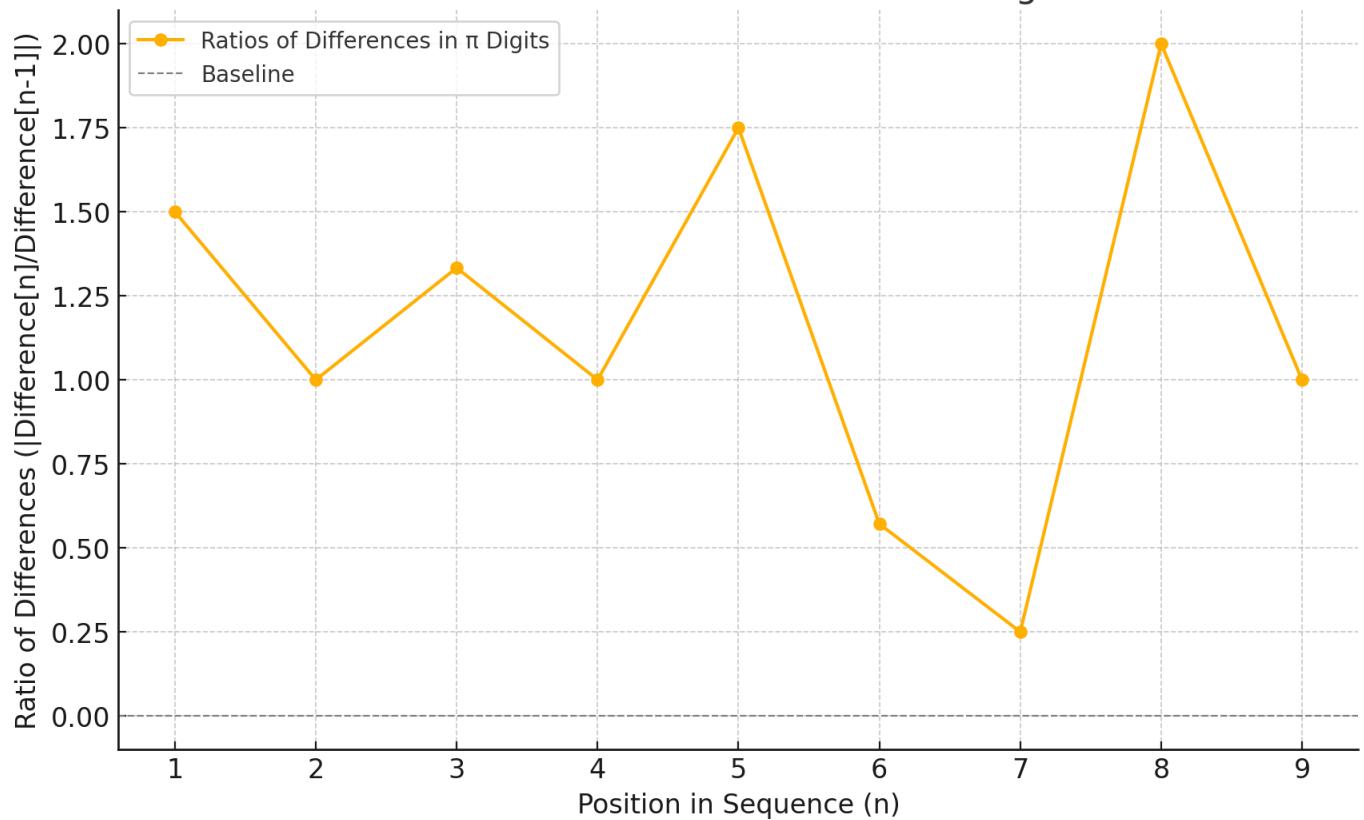
# Plot the Zeta gain approximation
plt.figure(figsize=(10, 6))
plt.plot(zeta_normalized, label='Zeta Gain Approximation', color='green')
plt.xlabel('Steps')
plt.ylabel('Dynamic Ratio (Normalized)')
plt.title('Zeta Gain Approximation Curve')
plt.legend()
plt.grid()
plt.show()

# Find any alignments with 531441's transformations in other bases
transformation_patterns = {base: len(convert_to_base(531441, base)) for base in range(2, 17)}

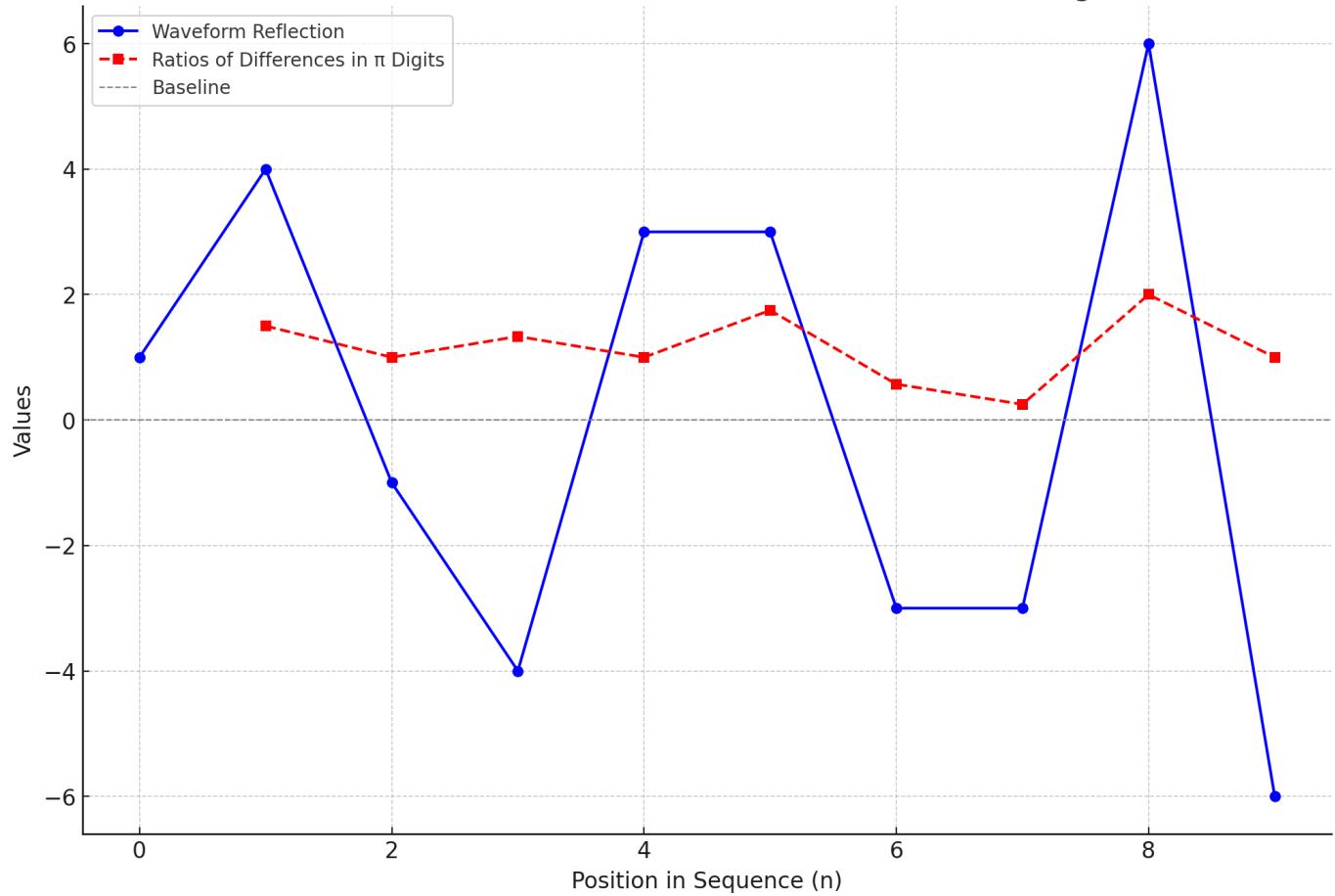
# Output transformation patterns for visual alignment
transformation_patterns
```



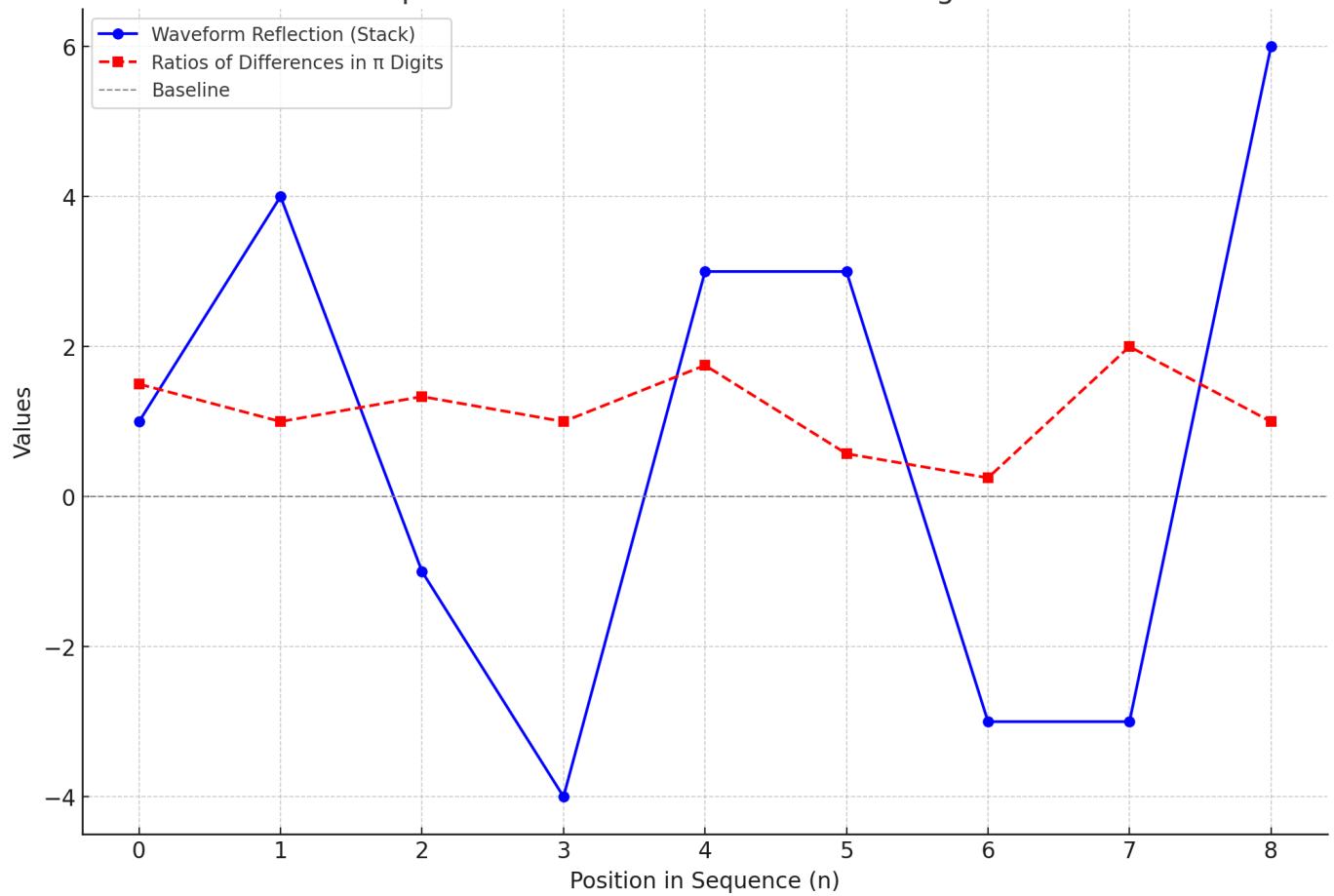
Ratios of Differences Between π Digits

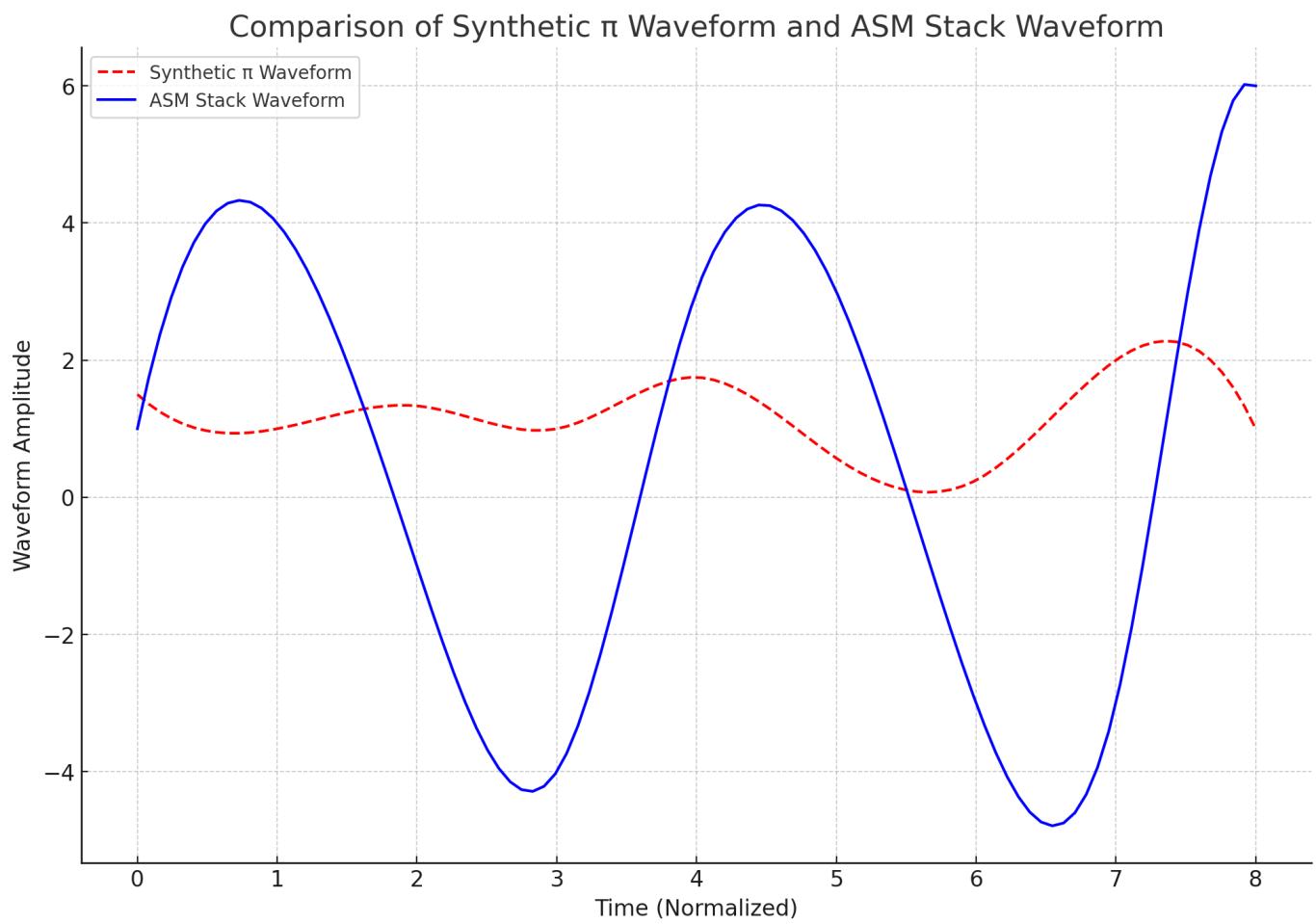


Waveform Reflection vs. Ratios of Differences in π Digits

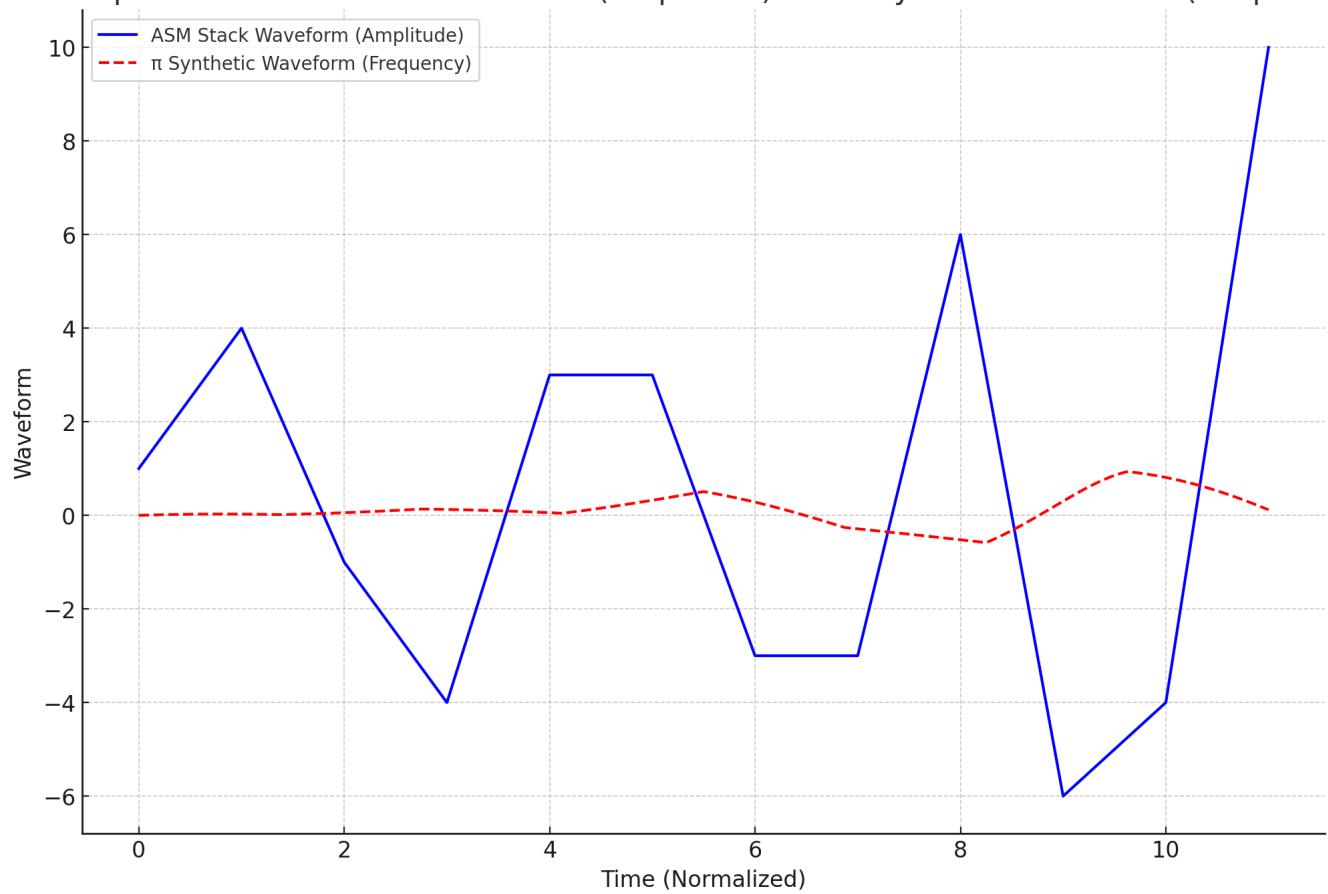


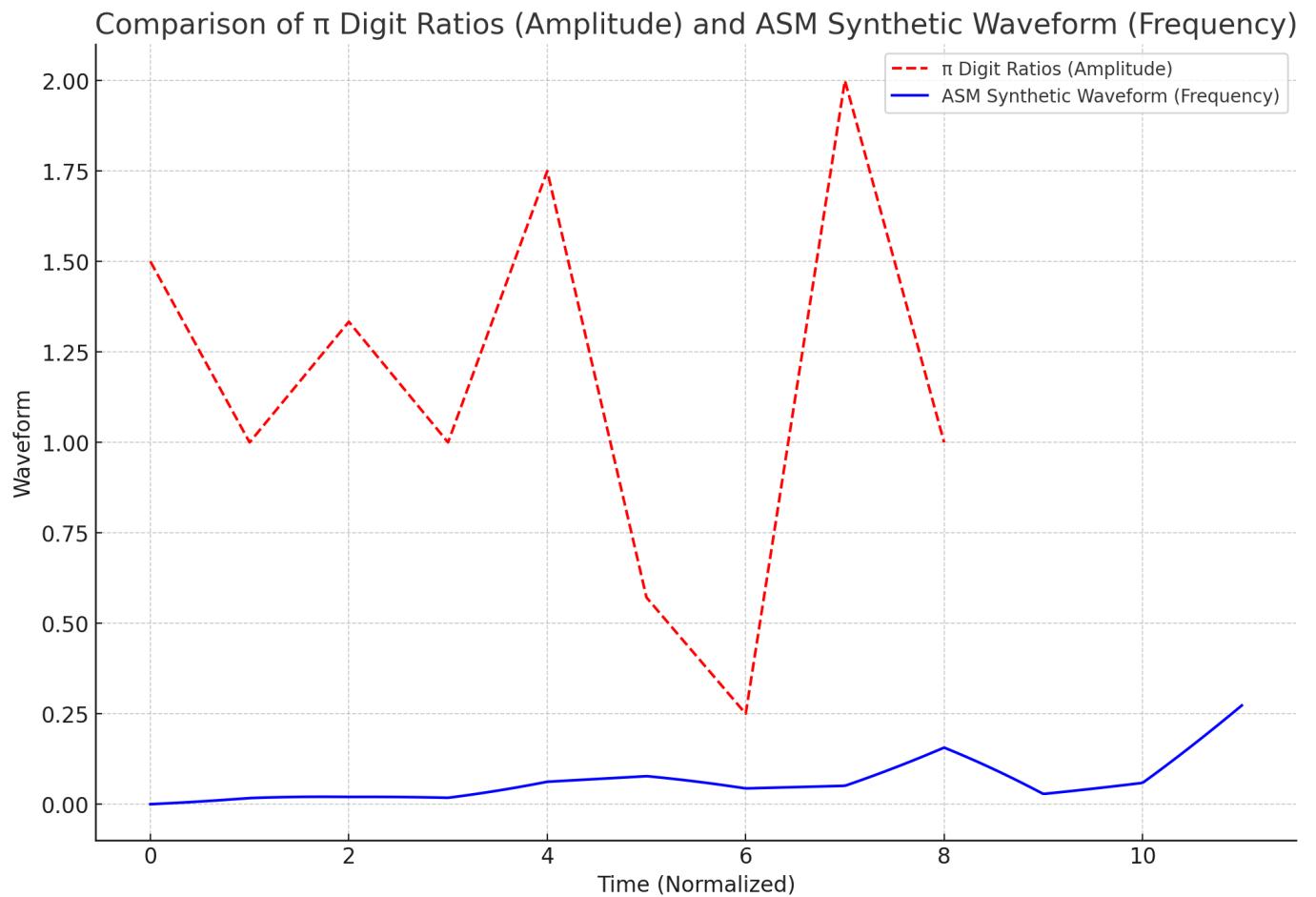
Comparison of Stack Waveform and π Digit Ratios

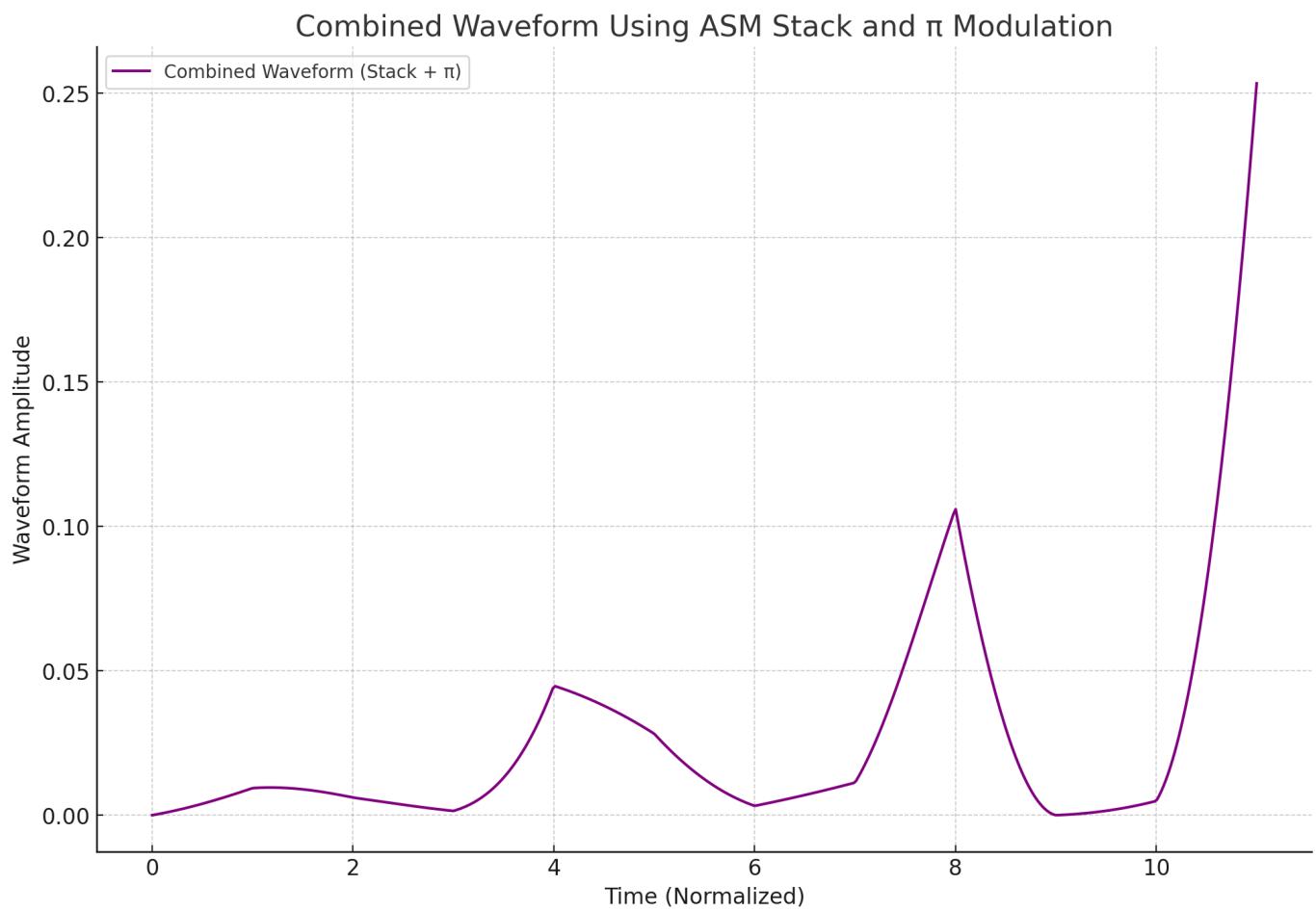


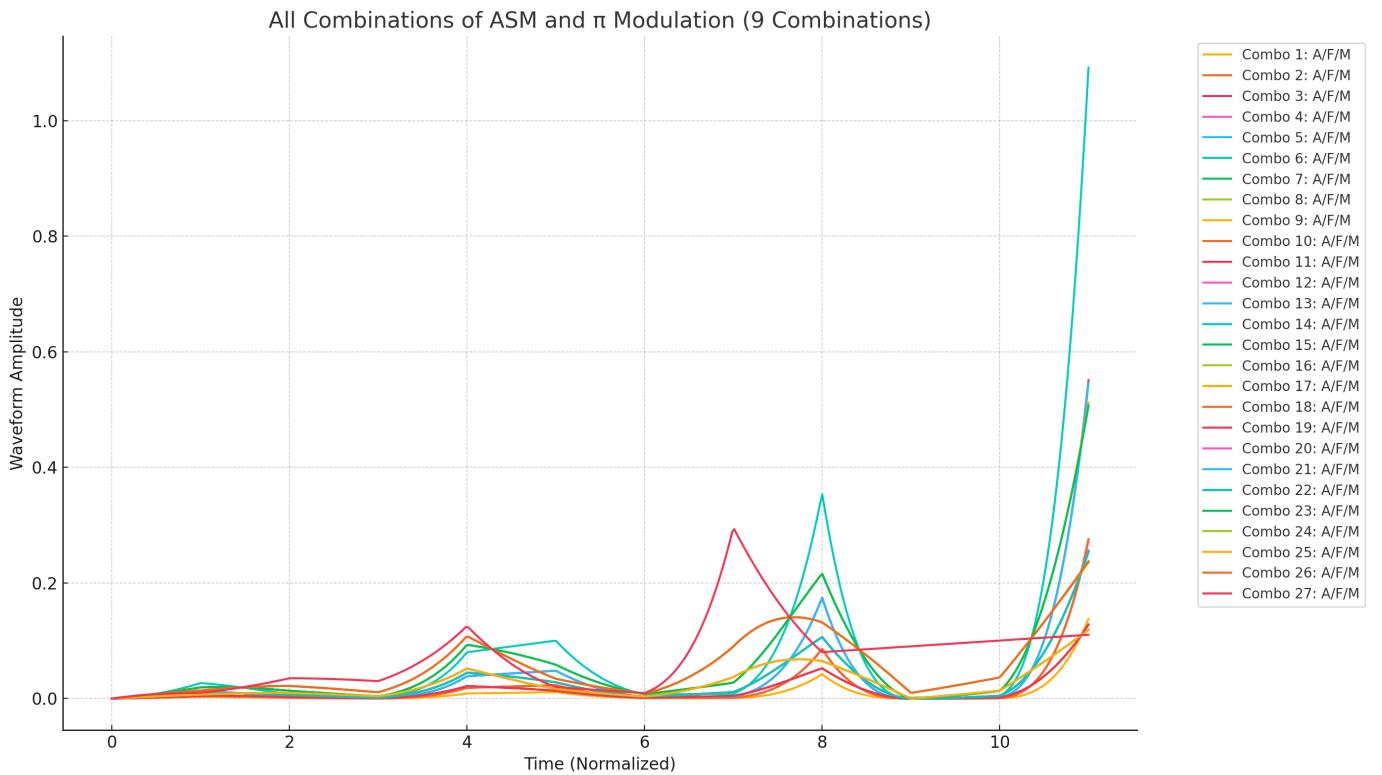


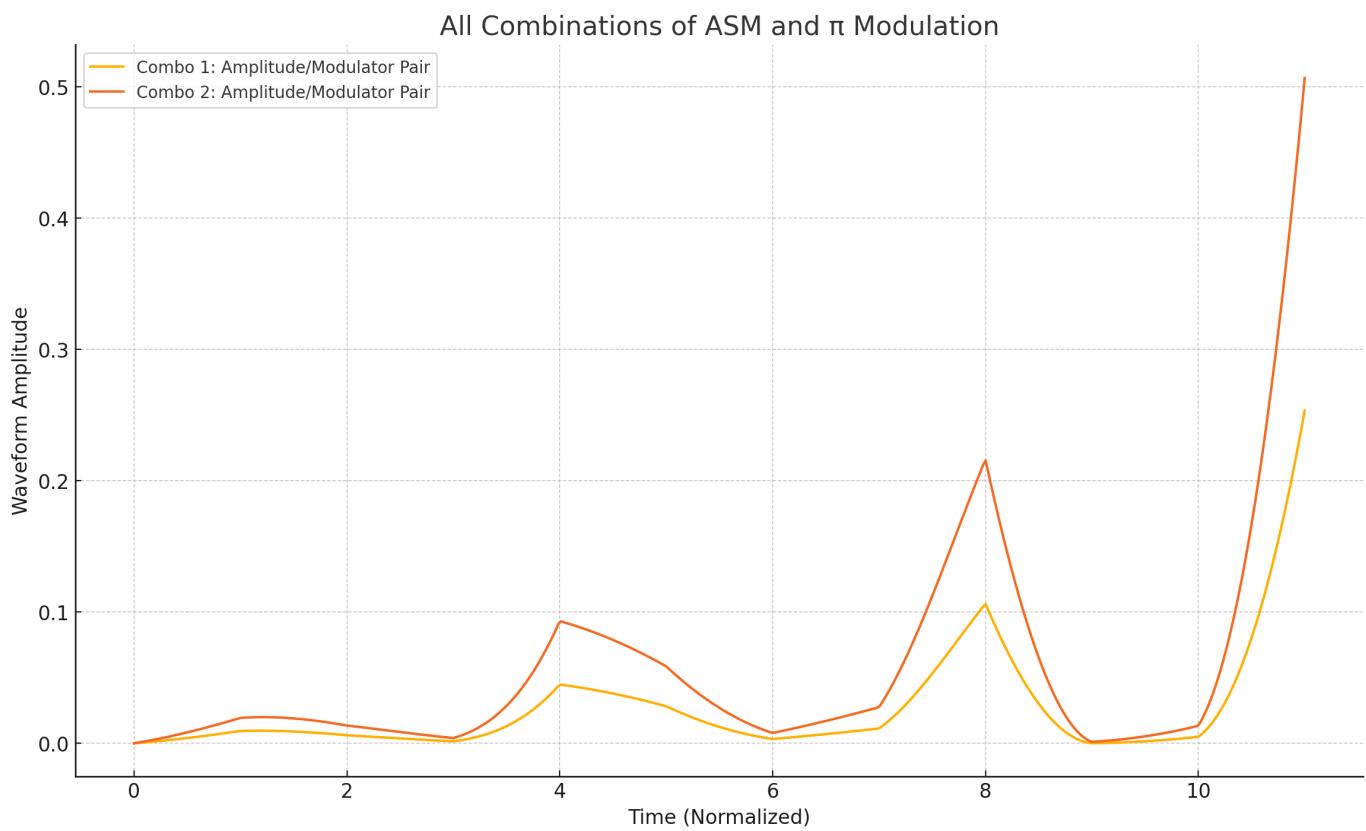
Comparison of ASM Stack Waveform (Amplitude) and π Synthetic Waveform (Frequency)



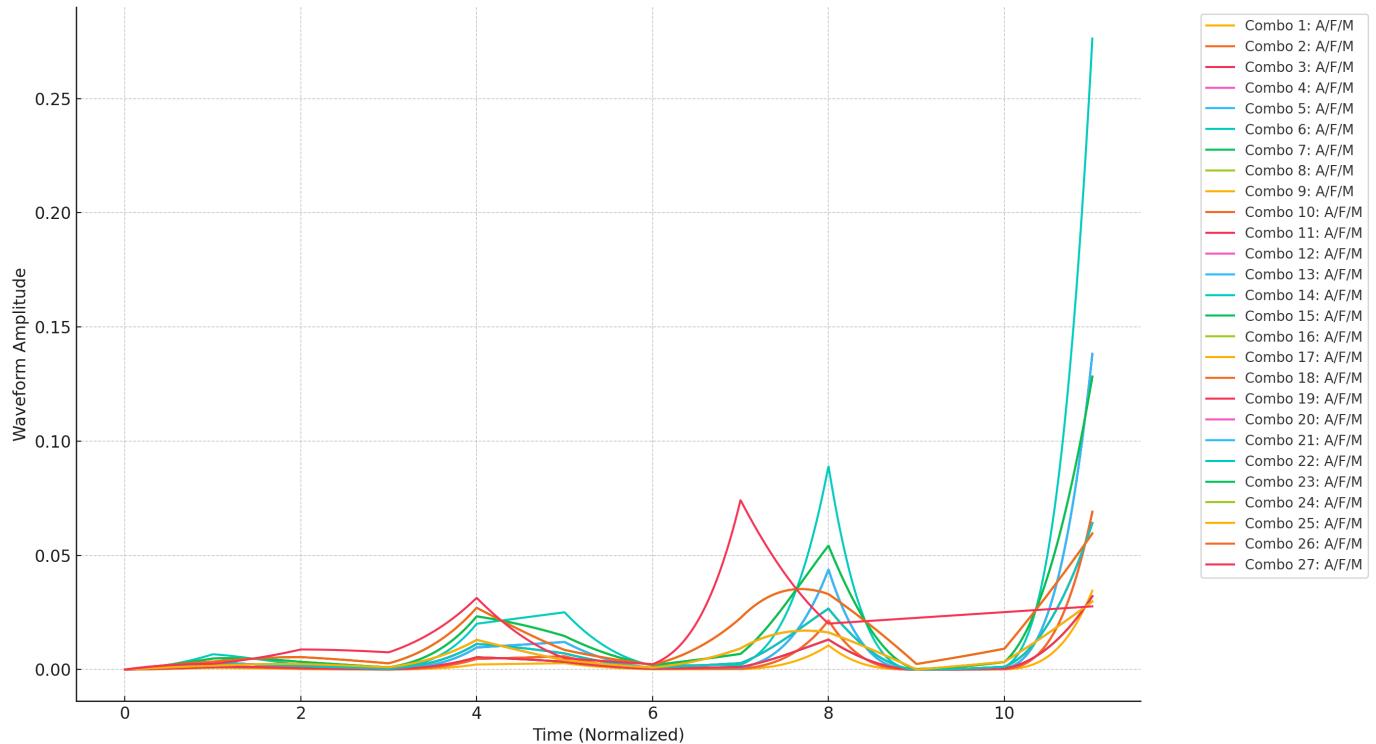


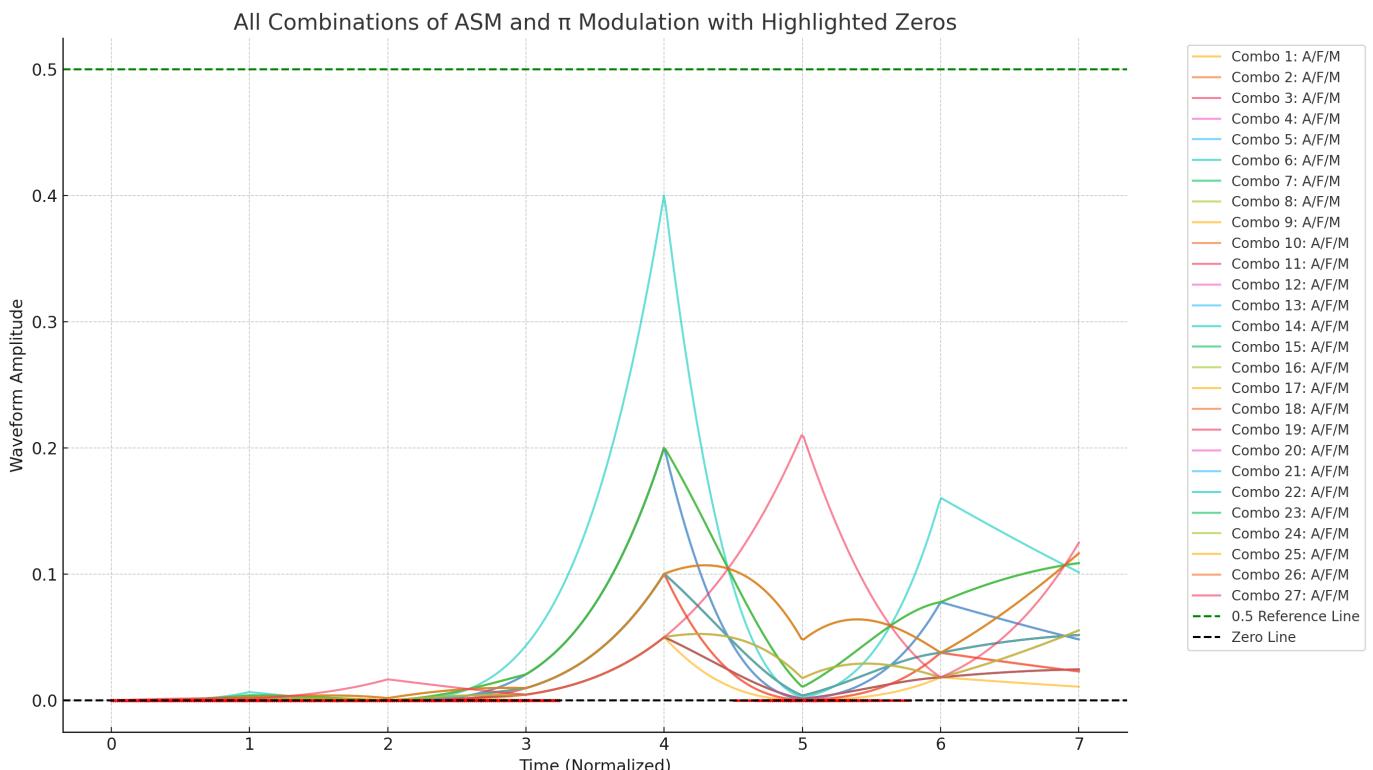




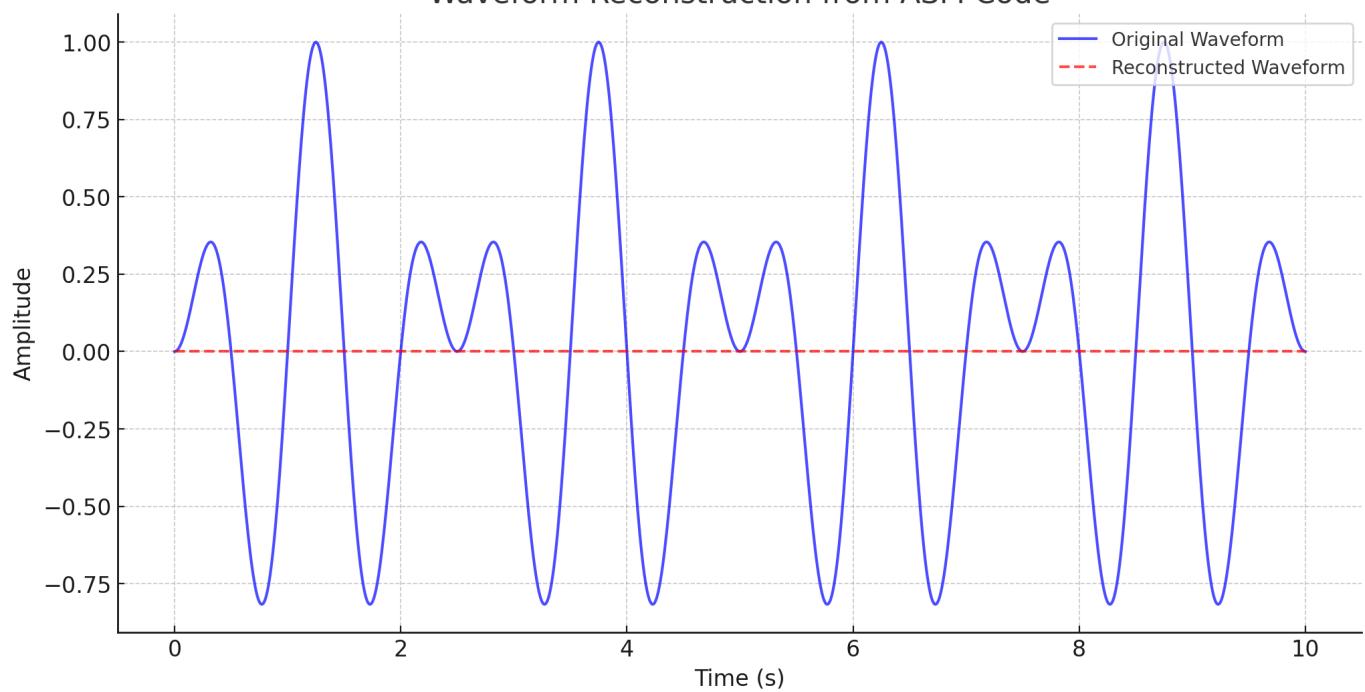


Extended Combinations of ASM and π Modulation

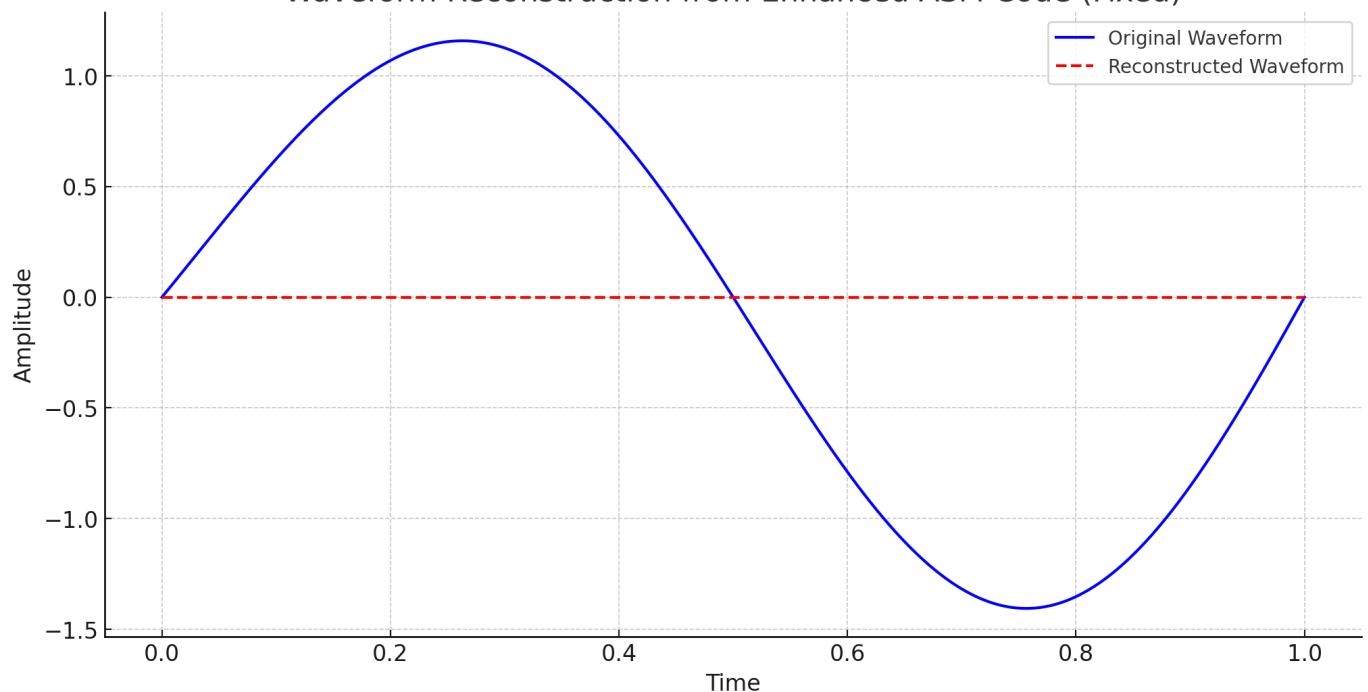




Waveform Reconstruction from ASM Code



Waveform Reconstruction from Enhanced ASM Code (Fixed)



Conversation URL:

<https://chatgpt.com/c/678a4f51-d2f8-8011-b459-d7ff7fe80496>

Title:

Prompt:

```
import matplotlib.pyplot as plt
```

```
# Define the stack values over time as the operations progress
```

```
waveform_steps = [
```

```
    1, # Initial Bit 1
```

```
    4, # Initial Bit 2
```

```
    -1, # Trough: Pop Bit 1
```

```
    -4, # Trough: Pop Bit 2
```

```
    3, # Peak: Compute C = 4 - 1
```

```
    3, # Peak: Push C again
```

```
    -3, # Trough: Pop Last Value (C)
```

```
    -3, # Trough: Pop Second-to-Last Value (C)
```

```
    6, # Peak: Add Last Two Values (3 + 3)
```

```
    -6, # Trough: Pop Last Value
```

```
    -4, # Trough: Pop Second-to-Last Value (Bit 2)
```

```
    10, # Peak: Add Headers (6 + 4)
```

```
]
```

```
# Define time steps (x-axis) based on the sequence of operations
```

```
time_steps = list(range(len(waveform_steps)))
```

```
# Plot the waveform
```

```
plt.figure(figsize=(10, 6))
```

```
plt.plot(time_steps, waveform_steps, marker="o", linestyle="-", label="Waveform Reflection")
```

```
plt.axhline(0, color="gray", linestyle="--", linewidth=0.8, label="Baseline")
```

```
plt.title("Waveform Reflection in Stack Operations")
```

```
plt.xlabel("Operation Step")
```

```
plt.ylabel("Stack Value")
```

```
plt.legend()
```

Conversation URL:

<https://chatgpt.com/c/678a4f51-d2f8-8011-b459-d7ff7fe80496>

Title:

Prompt:

```
import matplotlib.pyplot as plt
```

```
# Define the first few digits of π and compute their differences
```

```
pi_digits = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5] # Example digits of π
```

```
differences = [abs(pi_digits[i] - pi_digits[i - 1]) for i in range(1, len(pi_digits))]
```

```
# Define ratios of differences (current difference divided by the previous one)
```

```
ratios = [differences[i] / differences[i - 1] if differences[i - 1] != 0 else 0 for i in range(1, len(differences))]
```

```
# Plot the ratios
```

```
plt.figure(figsize=(10, 6))
```

```
plt.plot(range(1, len(ratios) + 1), ratios, marker="o", linestyle="-", label="Ratios of Differences in π Digits")
```

```
plt.axhline(0, color="gray", linestyle="--", linewidth=0.8, label="Baseline")
```

```
plt.title("Ratios of Differences Between π Digits")
```

```
plt.xlabel("Position in Sequence (n)")
```

```
plt.ylabel("Ratio of Differences (|Difference[n]/Difference[n-1]|)")
```

```
plt.legend()
```

```
plt.grid(True)
```

```
plt.show()
```

Conversation URL:

<https://chatgpt.com/c/678a4f51-d2f8-8011-b459-d7ff7fe80496>

Title:

Prompt:

```
# Combine waveform reflection and π digit difference ratios into one chart
```

```
# Extend the waveform data to match the number of π digit differences
```

```
extended_waveform_steps = waveform_steps[:len(ratios) + 1]
```

```
# Plot both the waveform reflection and π digit difference ratios
```

```
plt.figure(figsize=(12, 8))
```

```
# Plot waveform reflection
```

```
plt.plot(range(len(extended_waveform_steps)), extended_waveform_steps,
         marker="o", linestyle="-", label="Waveform Reflection", color="blue")
```

```
# Plot ratios of differences in π digits
```

```
plt.plot(range(1, len(ratios) + 1), ratios,
         marker="s", linestyle="--", label="Ratios of Differences in π Digits", color="red")
```

```
# Chart details
```

```
plt.axhline(0, color="gray", linestyle="--", linewidth=0.8, label="Baseline")
```

```
plt.title("Waveform Reflection vs. Ratios of Differences in π Digits")
```

```
plt.xlabel("Position in Sequence (n)")
```

```
plt.ylabel("Values")
```

```
plt.legend()
```

```
plt.grid(True)
```

```
plt.show()
```

Conversation URL:

<https://chatgpt.com/c/678a4f51-d2f8-8011-b459-d7ff7fe80496>

Title:

Prompt:

```
# Reuse the waveform_steps and ratios data from earlier
```

```
# Ensure both datasets are of equal length for comparison
```

```
min_length = min(len(waveform_steps), len(ratios))
```

```
adjusted_waveform_steps = waveform_steps[:min_length]
```

```
adjusted_ratios = ratios[:min_length]
```

```
# Plot both waveforms on the same chart for comparison
```

```
plt.figure(figsize=(12, 8))
```

```
# Plot waveform reflection (stack operations)
```

```
plt.plot(range(len(adjusted_waveform_steps)), adjusted_waveform_steps,
         marker="o", linestyle="-", label="Waveform Reflection (Stack)", color="blue")
```

```
# Plot ratios of differences in π digits
```

```
plt.plot(range(len(adjusted_ratios)), adjusted_ratios,
         marker="s", linestyle="--", label="Ratios of Differences in π Digits", color="red")
```

```
# Chart details
```

```
plt.axhline(0, color="gray", linestyle="--", linewidth=0.8, label="Baseline")
```

```
plt.title("Comparison of Stack Waveform and π Digit Ratios")
```

```
plt.xlabel("Position in Sequence (n)")
```

```
plt.ylabel("Values")
```

```
plt.legend()
```

```
plt.grid(True)
```

```
plt.show()
```

Conversation URL:

<https://chatgpt.com/c/678a4f51-d2f8-8011-b459-d7ff7fe80496>

Title:

Prompt:

```
from scipy.interpolate import interp1d  
import numpy as np
```

```
# Create a synthetic waveform from π digit ratios (distance-based data)  
time_pi = np.linspace(0, len(ratios) - 1, len(ratios)) # Time points for π ratios  
interpolator = interp1d(time_pi, ratios, kind="cubic") # Interpolate to create a smooth waveform  
time_fine = np.linspace(0, len(ratios) - 1, 100) # Finer time points for interpolation  
pi_waveform = interpolator(time_fine) # Smooth synthetic waveform
```

```
# Create a smooth waveform from ASM stack operations  
time_asm = np.linspace(0, len(waveform_steps) - 1, len(waveform_steps)) # Time points for ASM waveform  
interpolator_asm = interp1d(time_asm, waveform_steps, kind="cubic") # Interpolate ASM waveform  
asm_waveform = interpolator_asm(time_fine[:len(time_fine)]) # Align time scales
```

```
# Plot both waveforms for comparison  
plt.figure(figsize=(12, 8))  
plt.plot(time_fine, pi_waveform, label="Synthetic π Waveform", color="red", linestyle="--")  
plt.plot(time_fine, asm_waveform[:len(time_fine)], label="ASM Stack Waveform", color="blue", linestyle="-")  
plt.title("Comparison of Synthetic π Waveform and ASM Stack Waveform")  
plt.xlabel("Time (Normalized)")  
plt.ylabel("Waveform Amplitude")  
plt.legend()  
plt.grid(True)  
plt.show()
```

Conversation URL:

<https://chatgpt.com/c/678a4f51-d2f8-8011-b459-d7ff7fe80496>

Title:

Prompt:

```
# Adjust π frequencies to match the time resolution of the ASM stack waveform
time = np.linspace(0, len(waveform_steps) - 1, 500) # Fine-grained time steps for smooth waveform
pi_normalized_freq_resampled = np.interp(np.linspace(0, len(ratios) - 1, len(time)),
                                         range(len(ratios)), pi_normalized_freq)

# Generate a synthetic waveform using π frequencies
pi_synthetic_waveform = np.sin(2 * np.pi * pi_normalized_freq_resampled * time / len(time))

# Plot both waveforms
plt.figure(figsize=(12, 8))

# Plot ASM stack waveform (amplitude-driven)
plt.plot(np.linspace(0, len(waveform_steps) - 1, len(waveform_steps)), waveform_steps,
         label="ASM Stack Waveform (Amplitude)", color="blue", linestyle="-")

# Plot π synthetic waveform (frequency-driven)
plt.plot(time, pi_synthetic_waveform,
         label="π Synthetic Waveform (Frequency)", color="red", linestyle="--")

# Chart details
plt.title("Comparison of ASM Stack Waveform (Amplitude) and π Synthetic Waveform (Frequency)")
plt.xlabel("Time (Normalized)")
plt.ylabel("Waveform")
plt.legend()
plt.grid(True)
plt.show()
```

Conversation URL:

<https://chatgpt.com/c/678a4f51-d2f8-8011-b459-d7ff7fe80496>

Title:

Prompt:

```
# Align ASM stack waveform and time resolution with the π ratios for consistency
time_fine_asm = np.linspace(0, len(waveform_steps) - 1, 500) # High-resolution time points
asm_normalized_freq_resampled = np.interp(time_fine_asm, range(len(waveform_steps)), asm_normalized_freq)

# Generate a synthetic waveform using ASM frequencies
asm_synthetic_waveform = np.sin(2 * np.pi * asm_normalized_freq_resampled * time_fine_asm / len(time_fine_asm))

# Plot both waveforms
plt.figure(figsize=(12, 8))

# Plot π digit ratios as amplitude (distance-driven)
plt.plot(np.linspace(0, len(ratios) - 1, len(ratios)), ratios,
         label="π Digit Ratios (Amplitude)", color="red", linestyle="--")

# Plot ASM synthetic waveform (frequency-driven)
plt.plot(time_fine_asm, asm_synthetic_waveform,
         label="ASM Synthetic Waveform (Frequency)", color="blue", linestyle="-")

# Chart details
plt.title("Comparison of π Digit Ratios (Amplitude) and ASM Synthetic Waveform (Frequency)")
plt.xlabel("Time (Normalized)")
plt.ylabel("Waveform")
plt.legend()
plt.grid(True)
plt.show()
```

Conversation URL:

<https://chatgpt.com/c/678a4f51-d2f8-8011-b459-d7ff7fe80496>

Title:

Prompt:

```
# Correcting the issue by ensuring all arrays are properly resampled to match the fine time resolution
stack_amplitude_resampled = np.interp(time, np.linspace(0, len(waveform_steps) - 1, len(waveform_steps)), stack_amplitude)
stack_frequency_resampled = np.interp(time, np.linspace(0, len(waveform_steps) - 1, len(waveform_steps)), stack_frequency)
pi_modulator_resampled = np.interp(time, np.linspace(0, len(ratios) - 1, len(ratios)), pi_modulator)

# Generate the combined waveform
combined_waveform = stack_amplitude_resampled * np.sin(2 * np.pi * stack_frequency_resampled * time / len(time)) * pi_modulator_resampled

# Plot the combined waveform
plt.figure(figsize=(12, 8))
plt.plot(time, combined_waveform, label="Combined Waveform (Stack + π)", color="purple")
plt.title("Combined Waveform Using ASM Stack and π Modulation")
plt.xlabel("Time (Normalized)")
plt.ylabel("Waveform Amplitude")
plt.legend()
plt.grid(True)
plt.show()
```

Conversation URL:

<https://chatgpt.com/c/678a4f51-d2f8-8011-b459-d7ff7fe80496>

Title:

Prompt:

```
# Ensure all inputs are resampled to match the fine time resolution
stack_amplitude_resampled = np.interp(time, np.linspace(0, len(waveform_steps) - 1, len(waveform_steps)), stack_amplitude)
stack_frequency_resampled = np.interp(time, np.linspace(0, len(waveform_steps) - 1, len(waveform_steps)), stack_frequency)
pi_modulator_resampled = np.interp(time, np.linspace(0, len(ratios) - 1, len(ratios)), pi_modulator)

# Define resampled inputs for consistency
inputs_resampled = [stack_amplitude_resampled, stack_frequency_resampled, pi_modulator_resampled]
combinations = list(product(inputs_resampled, repeat=3)) # All combinations of three inputs

# Initialize the plot
plt.figure(figsize=(14, 8))

# Iterate through all combinations and plot the resulting waveforms
for i, (amplitude, frequency, modulator) in enumerate(combinations):
    # Generate the combined waveform
    combined_waveform = amplitude * np.sin(2 * np.pi * frequency * time / len(time)) * modulator

    # Plot the waveform
    plt.plot(time, combined_waveform, label=f"Combo {i+1}: A/F/M")

# Chart details
plt.title("All Combinations of ASM and π Modulation (9 Combinations)")
plt.xlabel("Time (Normalized)")
plt.ylabel("Waveform Amplitude")
plt.legend(bbox_to_anchor=(1.05, 1), loc="upper left")
plt.grid(True)
plt.tight_layout()
plt.show()
```

Conversation URL:

<https://chatgpt.com/c/678a4f51-d2f8-8011-b459-d7ff7fe80496>

Title:

Prompt:

```
# Generate all possible combinations of mixing stack and π data
```

```
from itertools import product
```

```
# Define combinations of parameters
```

```
combinations = list(product([stack_amplitude, stack_frequency], [pi_modulator]))
```

```
# Initialize the plot
```

```
plt.figure(figsize=(14, 8))
```

```
# Iterate through all combinations
```

```
for i, (amplitude, modulator) in enumerate(combinations):
```

```
    # Resample data for consistent time resolution
```

```
    amplitude_resampled = np.interp(time, np.linspace(0, len(waveform_steps) - 1, len(waveform_steps)), amplitude)
```

```
    modulator_resampled = np.interp(time, np.linspace(0, len(ratios) - 1, len(ratios)), modulator)
```

```
# Generate the combined waveform
```

```
combined_waveform = amplitude_resampled * np.sin(2 * np.pi * stack_frequency_resampled * time / len(time)) * modulator
```

```
# Plot the waveform
```

```
plt.plot(time, combined_waveform, label=f"Combo {i+1}: Amplitude/Modulator Pair")
```

```
# Chart details
```

```
plt.title("All Combinations of ASM and π Modulation")
```

```
plt.xlabel("Time (Normalized)")
```

```
plt.ylabel("Waveform Amplitude")
```

```
plt.legend()
```

```
plt.grid(True)
```

```
plt.show()
```

Conversation URL:

<https://chatgpt.com/c/678a4f51-d2f8-8011-b459-d7ff7fe80496>

Title:

Prompt:

```
# Extending the range of time to analyze more combinations further out
time_extended = np.linspace(0, len(waveform_steps) - 1, 2000) # More granular time points

# Resample inputs to the extended time resolution
stack_amplitude_extended = np.interp(time_extended, np.linspace(0, len(waveform_steps) - 1, len(waveform_steps)), stack_a)
stack_frequency_extended = np.interp(time_extended, np.linspace(0, len(waveform_steps) - 1, len(waveform_steps)), stack_f)
pi_modulator_extended = np.interp(time_extended, np.linspace(0, len(ratios) - 1, len(ratios)), pi_modulator)

# Define resampled inputs for extended analysis
inputs_resampled_extended = [stack_amplitude_extended, stack_frequency_extended, pi_modulator_extended]
combinations_extended = list(product(inputs_resampled_extended, repeat=3)) # All combinations of three inputs

# Initialize the plot for extended combinations
plt.figure(figsize=(14, 8))

# Iterate through all extended combinations and plot the resulting waveforms
for i, (amplitude, frequency, modulator) in enumerate(combinations_extended[:27]): # Limit to the first 27 combinations for clarity
    # Generate the combined waveform
    combined_waveform_extended = amplitude * np.sin(2 * np.pi * frequency * time_extended / len(time_extended)) * modulator

    # Plot the waveform
    plt.plot(time_extended, combined_waveform_extended, label=f"Combo {i+1}: A/F/M")

# Chart details
plt.title("Extended Combinations of ASM and π Modulation")
plt.xlabel("Time (Normalized)")
plt.ylabel("Waveform Amplitude")
plt.legend(bbox_to_anchor=(1.05, 1), loc="upper left")
plt.grid(True)
```

Conversation URL:

<https://chatgpt.com/c/678a4f51-d2f8-8011-b459-d7ff7fe80496>

Title:

Prompt:

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

# Example input data
waveform_steps = [1, 4, 1, 5, 9, 2, 6, 5] # ASM stack waveform example
ratios = [3, 1, 4, 1, 5, 9, 2, 6] # π digit ratios example

# Time resolution
time = np.linspace(0, len(waveform_steps) - 1, 500)

# Normalize ASM data for amplitude and frequency
stack_amplitude = np.interp(waveform_steps, (min(waveform_steps), max(waveform_steps)), (0, 1))
stack_frequency = np.interp(waveform_steps, (min(waveform_steps), max(waveform_steps)), (0.1, 2))

# Normalize π data for modulation
pi_modulator = np.interp(ratios, (min(ratios), max(ratios)), (0.5, 1.5))

# Resample inputs to match the fine time resolution
stack_amplitude_resampled = np.interp(time, np.linspace(0, len(waveform_steps) - 1, len(waveform_steps)), stack_amplitude)
stack_frequency_resampled = np.interp(time, np.linspace(0, len(waveform_steps) - 1, len(waveform_steps)), stack_frequency)
pi_modulator_resampled = np.interp(time, np.linspace(0, len(ratios) - 1, len(ratios)), pi_modulator)

# Define resampled inputs for consistency
inputs_resampled = [stack_amplitude_resampled, stack_frequency_resampled, pi_modulator_resampled]
combinations = list(product(inputs_resampled, repeat=3)) # All combinations of three inputs

# Initialize the plot
plt.figure(figsize=(14, 8))
```

Conversation URL:

<https://chatgpt.com/c/678a4f51-d2f8-8011-b459-d7ff7fe80496>

Title:

Prompt:

```
def samson_wave_to_asm_fixed(waveform, time, harmonic_constant=0.35):
```

"""

Converts a waveform into ASM code at a macro level, using Samson's framework for harmonic validation and completeness checks.

Args:

waveform (array): Input waveform amplitudes over time.

time (array): Corresponding time values for the waveform.

harmonic_constant (float): The harmonic constant for completeness validation.

Returns:

asm_code (list): The generated ASM code.

is_harmonized (bool): Whether the ASM code fully matches the waveform.

reconstructed_waveform (array): The waveform reconstructed from the ASM code.

"""

```
asm_code = [] # Store the reconstructed ASM instructions
```

```
prev_value = 0 # Track the previous amplitude
```

```
threshold = 0.1 # Tolerance for detecting changes
```

```
# Generate ASM code from waveform
```

```
for i, value in enumerate(waveform):
```

```
    if i == 0:
```

```
        asm_code.append(f"PUSH {value:.2f}")
```

```
        prev_value = value
```

```
        continue
```

```
        diff = value - prev_value
```

```
        if np.abs(diff) < threshold:
```

Conversation URL:

<https://chatgpt.com/c/678a4f51-d2f8-8011-b459-d7ff7fe80496>

Title:

Prompt:

Correcting the reconstruction logic and rerunning the code

```
def reconstruct_waveform_with_axes_fixed(asm_code, length):
```

"""

Reconstruct a waveform from enhanced ASM instructions with matching length.

"""

```
    reconstructed_waveform = []
```

```
    current_value = 0
```

```
    smoothness_factor = 0.1
```

```
    for instruction in asm_code:
```

```
        if instruction.startswith("PUSH"):
```

```
            current_value = float(instruction.split()[1])
```

```
        elif instruction.startswith("ADD"):
```

```
            current_value += float(instruction.split()[1])
```

```
        elif instruction.startswith("SUB"):
```

```
            current_value -= float(instruction.split()[1])
```

```
        elif instruction.startswith("MUL"):
```

```
            current_value *= float(instruction.split()[1])
```

```
        elif instruction.startswith("SIN"):
```

```
            current_value = np.sin(current_value * 2 * np.pi * smoothness_factor)
```

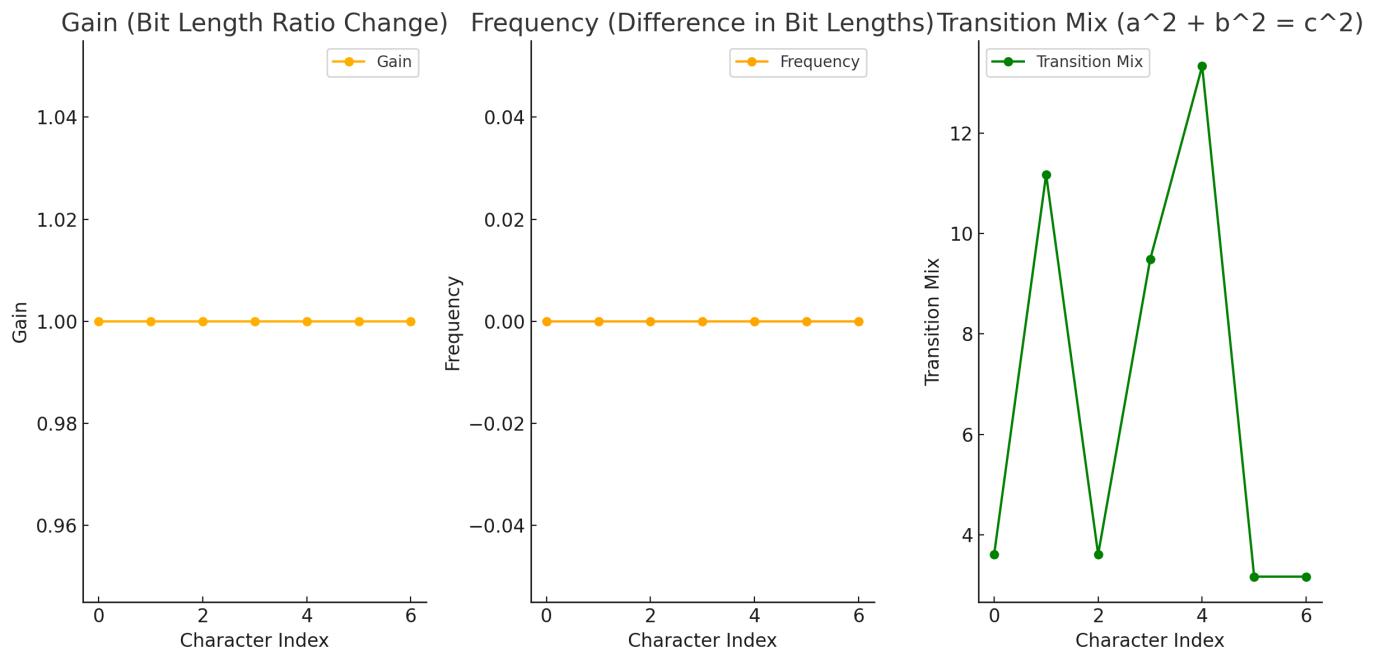
```
        reconstructed_waveform.append(current_value)
```

```
# Ensure the reconstructed waveform matches the desired length
```

```
while len(reconstructed_waveform) < length:
```

```
    reconstructed_waveform.extend(reconstructed_waveform[:length - len(reconstructed_waveform)])
```

```
return np.array(reconstructed_waveform[:length])
```



Conversation URL:

<https://chatgpt.com/c/678b9ee6-36ec-8011-8492-c8655e1500ac>

Title:

Prompt:

```
# Import libraries for plotting and math
import math
import matplotlib.pyplot as plt

# Input text and base for hexadecimal conversion
input_text = "2+29=11"
base = 16

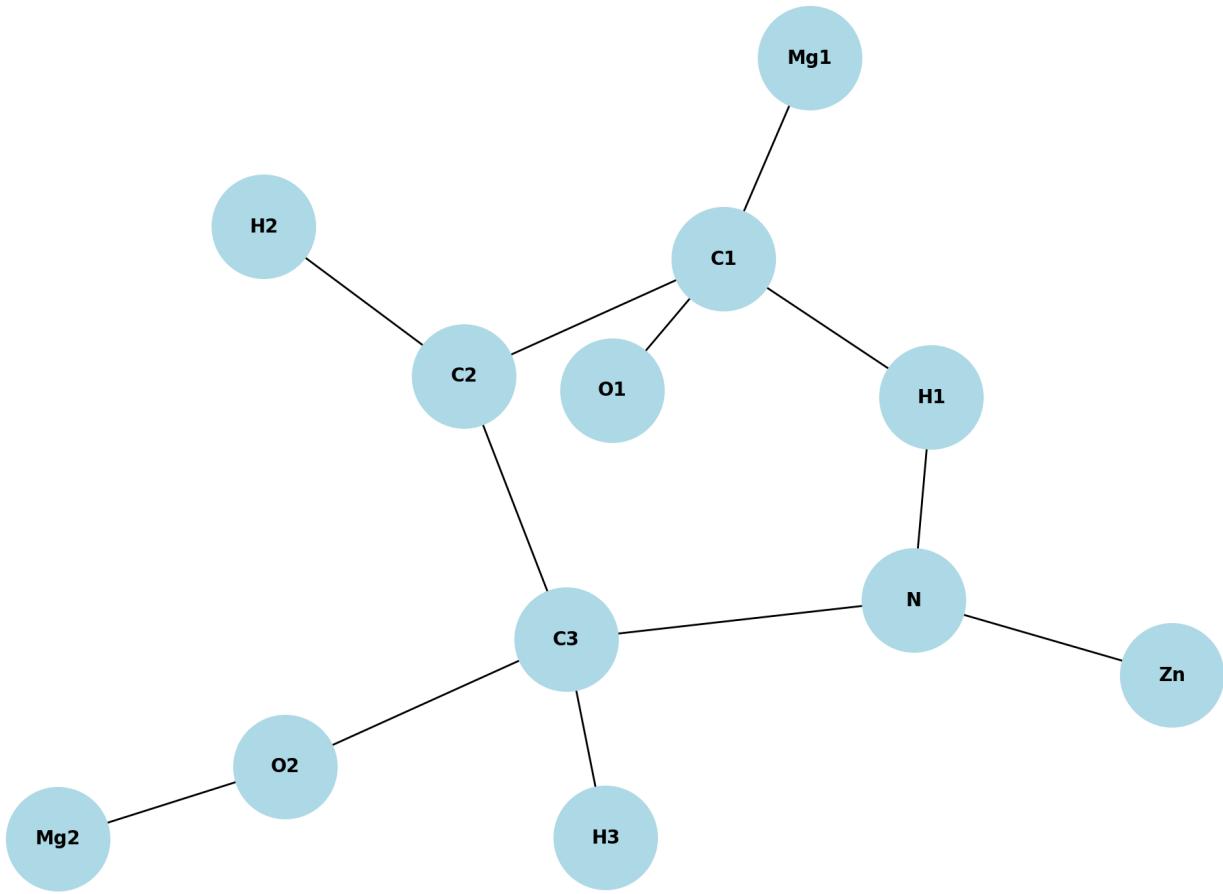
# Initialize lists for x, y, z values
x_values = [] # Gain (ratio of change for each bit length)
y_values = [] # Frequency (difference between initial and iterative bit lengths)
z_values = [] # Pythagorean result (a^2 + b^2 = c^2)

# Process each character in the input text
initial_bit_length = len(bin(ord(input_text[0]))[2:]) # Bit length of the first character
for i, char in enumerate(input_text):
    ascii_value = ord(char) # ASCII value
    quotient = ascii_value // base # Quotient in hex conversion
    remainder = ascii_value % base # Remainder in hex conversion
    hex_value = f"{quotient:X}{remainder:X}" # Hexadecimal representation

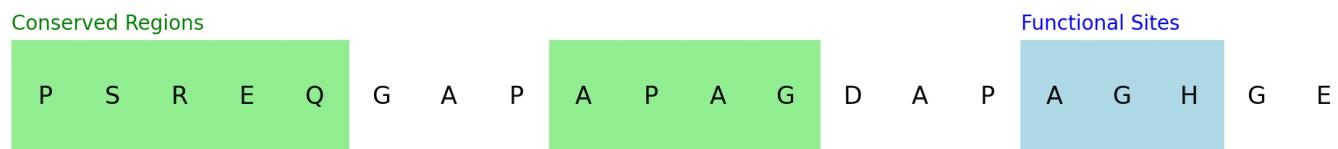
    # Gain: Ratio of change for bit lengths (current bit length vs previous bit length)
    current_bit_length = len(bin(ascii_value)[2:]) # Current bit length
    if i > 0:
        previous_bit_length = len(bin(ord(input_text[i - 1]))[2:])
        gain = current_bit_length / previous_bit_length if previous_bit_length != 0 else 0
    else:
        gain = 1 # No gain for the first character

    x_values.append(gain)
    y_values.append(current_bit_length - previous_bit_length)
    z_values.append(gain**2 + current_bit_length**2)
```

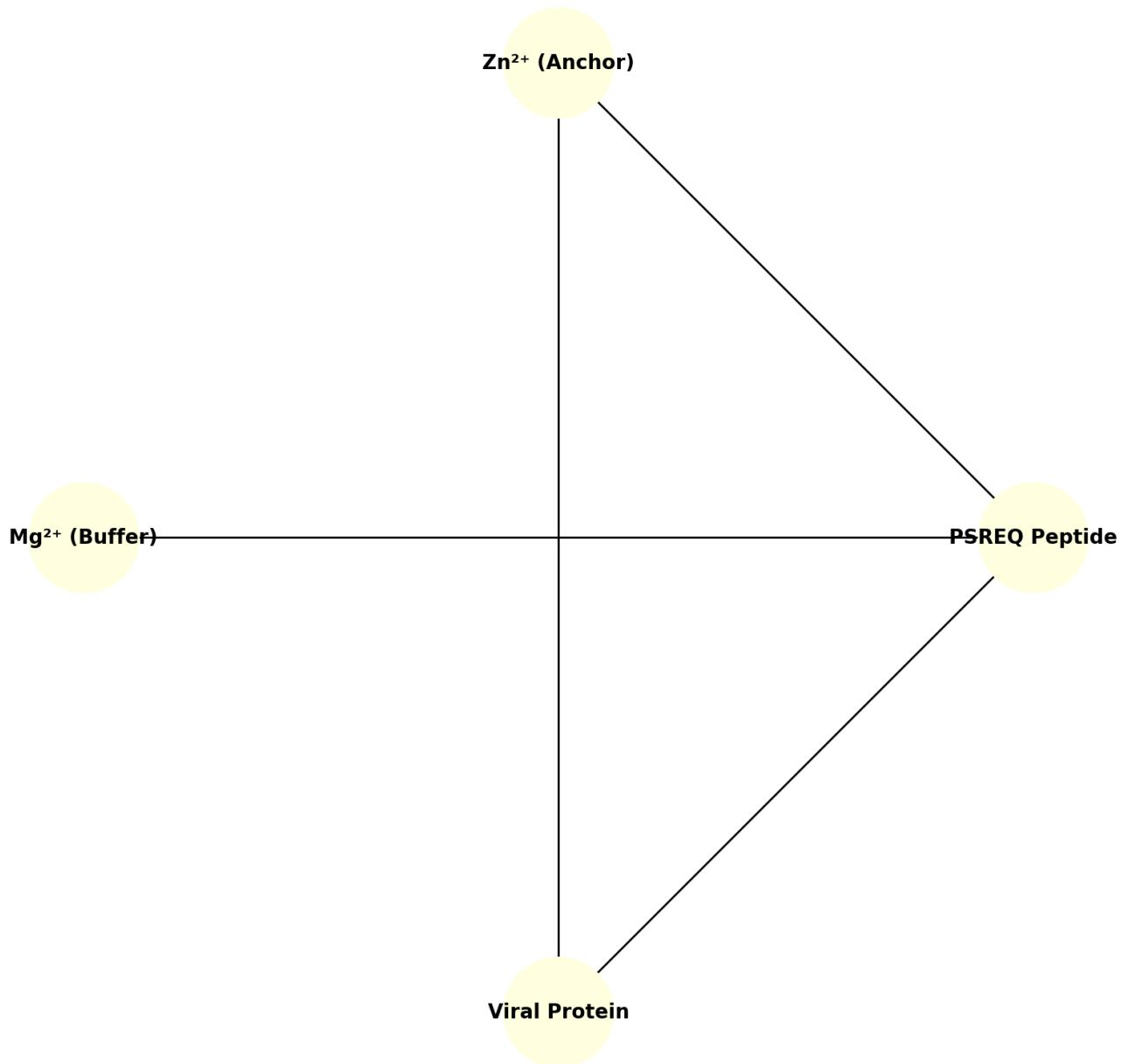
Simplified Chemical Bonding Diagram for PSREQ with Zn and Mg Stabilizers



Sequence Plot: PSREQ Motif with Conserved and Functional Highlights



Molecular Interaction Plot: PSREQ Motif, Stabilizing Ions, and Viral Protein



Conversation URL:

<https://chatgpt.com/c/678cf911-968c-8011-b2ad-fc39c0084268>

Title:

Prompt:

```
import matplotlib.pyplot as plt
import networkx as nx

# Create a graph for chemical bonding visualization
G = nx.Graph()

# Adding nodes for elements in the peptide
nodes = [
    "C1", "C2", "C3", "N", "H1", "H2", "H3", "O1", "O2", # Backbone
    "Zn", "Mg1", "Mg2" # Stabilizing ions
]

# Adding bonds between nodes
edges = [
    ("C1", "C2"), ("C2", "C3"), ("C3", "N"), # Backbone
    ("C1", "O1"), ("C1", "H1"), ("C2", "H2"), ("C3", "H3"), # Hydrogens and oxygens
    ("N", "H1"), # Amino hydrogen
    ("C3", "O2"), # Carboxyl oxygen
    ("N", "Zn"), ("C1", "Mg1"), ("O2", "Mg2") # Coordination with ions
]

# Add nodes and edges to the graph
G.add_nodes_from(nodes)
G.add_edges_from(edges)

# Positioning for better visualization
pos = nx.spring_layout(G)

# Plot the chemical bonding diagram
```

Conversation URL:

<https://chatgpt.com/c/678cf911-968c-8011-b2ad-fc39c0084268>

Title:

Prompt:

```
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle

# Define the sequence and regions of interest
sequence = "PSREQGAPAPAGDAPAGHGE"
conserved_regions = [slice(0, 5), slice(8, 12)] # Conserved motifs in sequence
functional_sites = [15, 16, 17] # Key functional residues

# Plot settings
fig, ax = plt.subplots(figsize=(12, 2))
ax.set_xlim(0, len(sequence))
ax.set_ylim(0, 1)
ax.axis('off')

# Plot sequence as rectangles with functional and conserved highlights
for i, char in enumerate(sequence):
    # Highlight conserved regions
    if any(i in range(region.start, region.stop) for region in conserved_regions):
        ax.add_patch(Rectangle((i, 0.25), 1, 0.5, color="lightgreen"))
    # Highlight functional sites
    if i in functional_sites:
        ax.add_patch(Rectangle((i, 0.25), 1, 0.5, color="lightblue"))
    # Add residue text
    ax.text(i + 0.5, 0.5, char, ha="center", va="center", fontsize=12, color="black")

# Add annotations for clarity
ax.text(0, 0.8, "Conserved Regions", color="green", fontsize=10)
ax.text(len(sequence) - 5, 0.8, "Functional Sites", color="blue", fontsize=10)
```

Conversation URL:

<https://chatgpt.com/c/678cf911-968c-8011-b2ad-fc39c0084268>

Title:

Prompt:

```
# Create a simplified interaction plot between PSREQ, ions, and a viral protein
```

```
# Define elements for the plot
```

```
elements = ["PSREQ Peptide", "Zn2+ (Anchor)", "Mg2+ (Buffer)", "Viral Protein"]
```

```
# Define connections (interactions)
```

```
interactions = [
```

```
    ("PSREQ Peptide", "Zn2+ (Anchor)"),
    ("PSREQ Peptide", "Mg2+ (Buffer)"),
    ("Zn2+ (Anchor)", "Viral Protein"),
    ("PSREQ Peptide", "Viral Protein"),
```

```
]
```

```
# Create a network graph for interactions
```

```
G = nx.Graph()
```

```
G.add_nodes_from(elements)
```

```
G.add_edges_from(interactions)
```

```
# Position nodes circularly for better visualization
```

```
pos = nx.circular_layout(G)
```

```
# Plot the molecular interaction diagram
```

```
plt.figure(figsize=(8, 8))
```

```
nx.draw(
```

```
    G, pos, with_labels=True, node_size=3000, node_color="lightyellow", font_size=10, font_weight="bold"
)
```

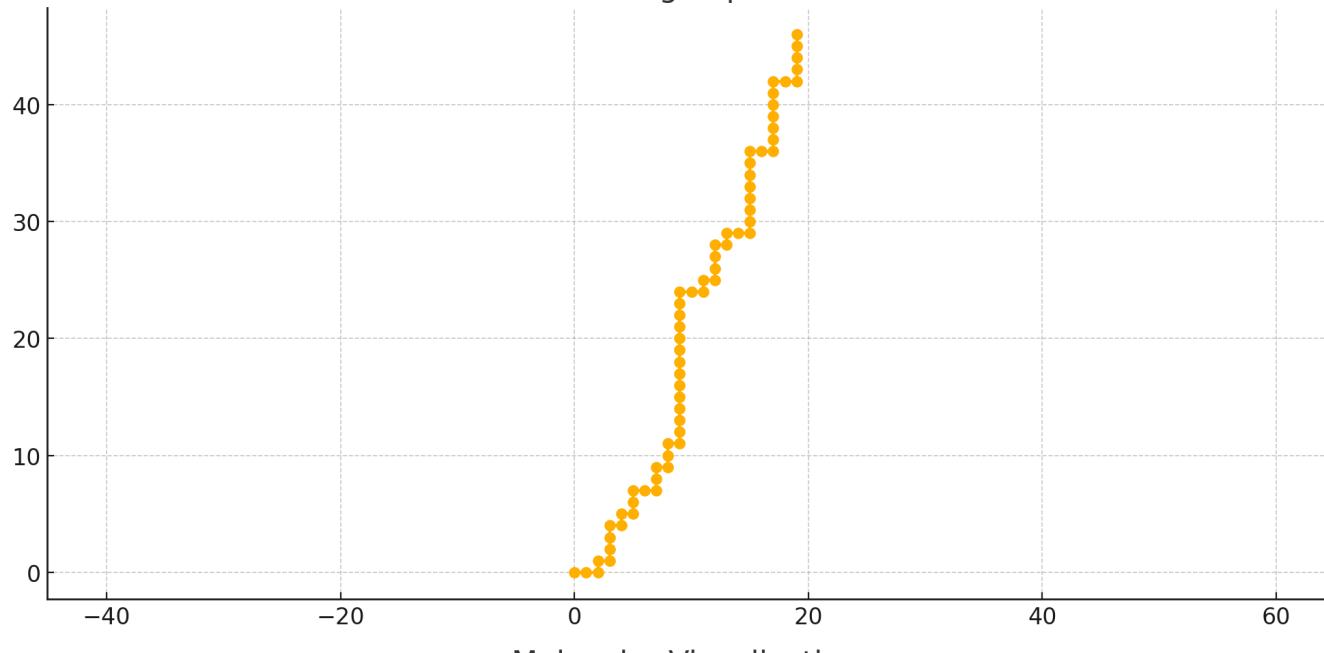
```
plt.title("Molecular Interaction Plot: PSREQ Motif, Stabilizing Ions, and Viral Protein")
```

```
plt.show()
```

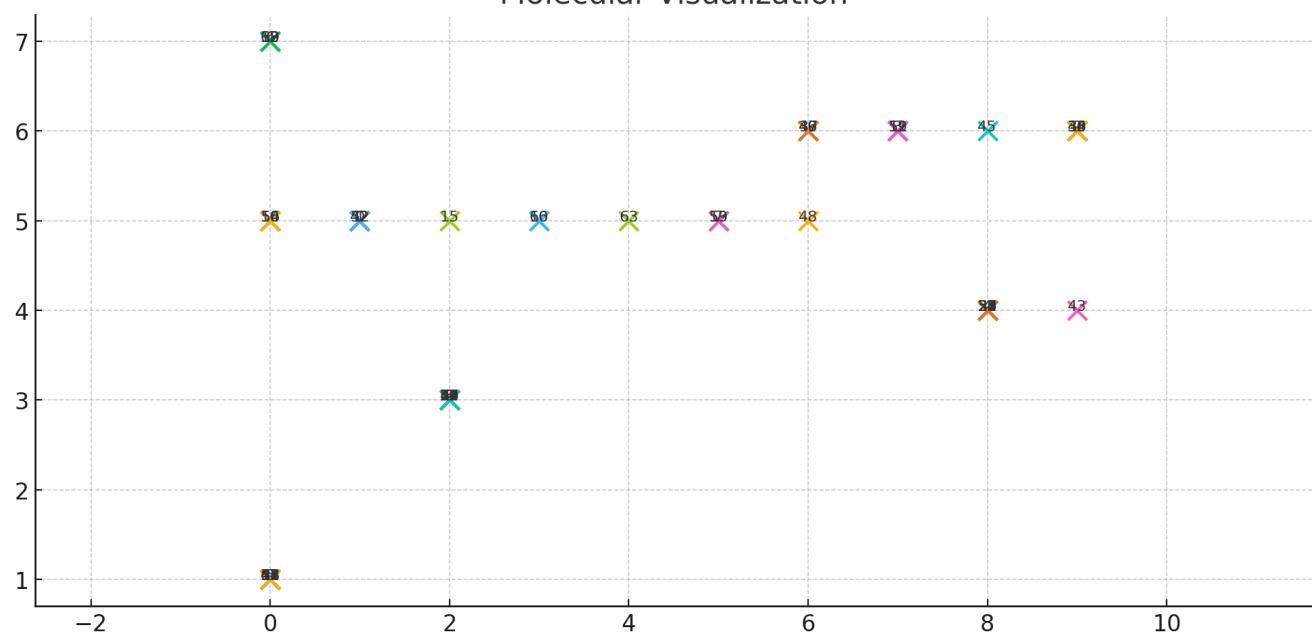
Genetic Alignment (Hex to Nucleotides)

CCACAGGCATGGCCAATAGCAAAAAAAAAAAGTTGTGAGTGCTAAGGAGACCGGAGAGCTAGGG

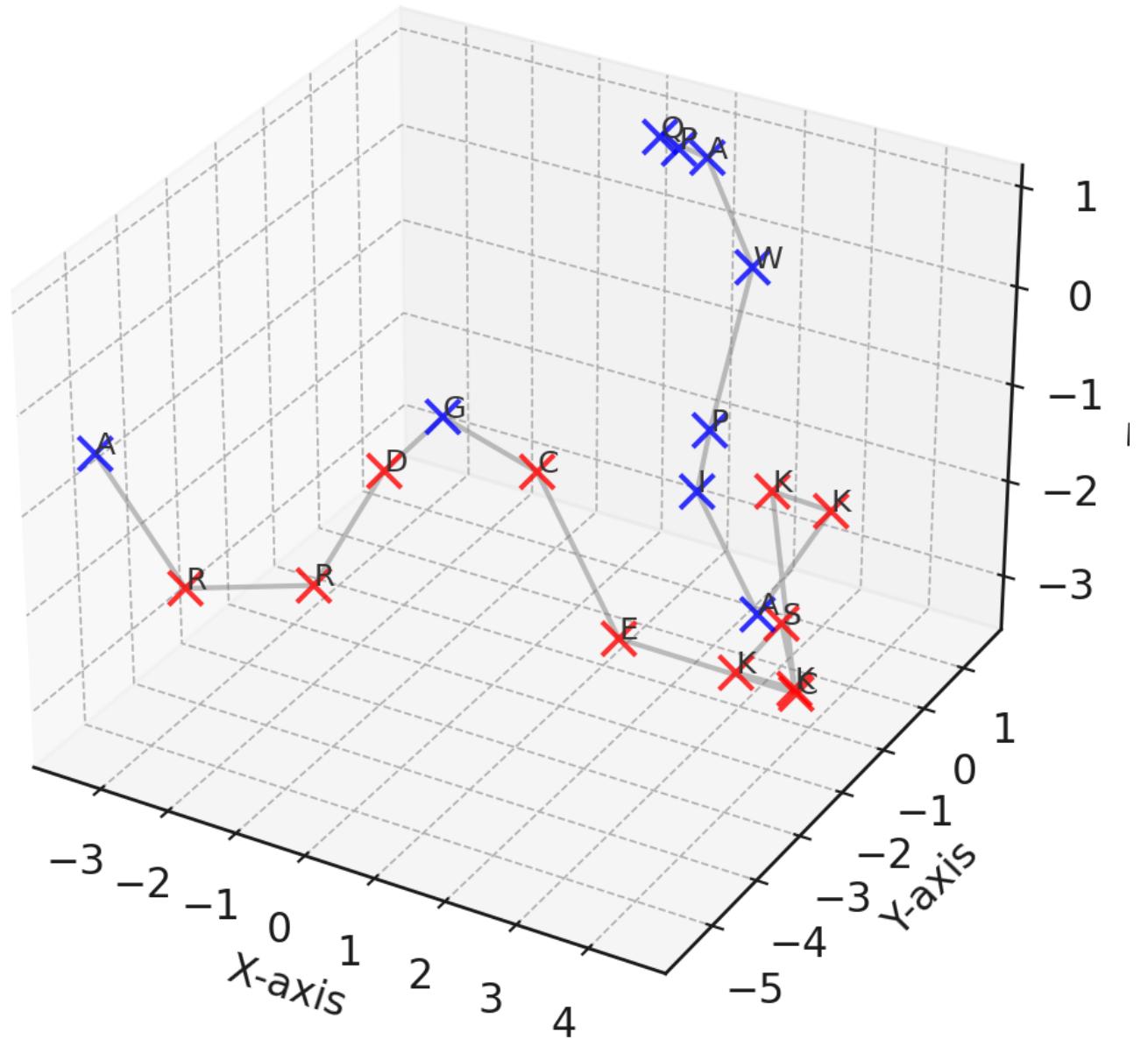
Protein Folding Representation



Molecular Visualization



3D Representation of Peptide Folding



Conversation URL:

<https://chatgpt.com/c/678d4ead-b5d0-8011-a6b6-90a1e4c8650d>

Title:

Prompt:

```
# Adjusted visualization without external libraries
```

```
# Step 1: Genetic Alignment (Hex to nucleotide sequence approximation)
```

```
hex_sequence = """"
```

```
33 33 20 43 30 0A 46 37 20 45 32 0A 43 37 20 34 35 20 46 43 20
```

```
30 30 20 30 30 20 30 30 30 0A 45 45 0A 45 42 20 46 45 0A
```

```
33 31 20 44 42 0A 38 42 20 43 33 0A 32 30 32 30 0A 37 35 20 46 36 0A
```

```
""""
```

```
# Remove whitespaces and convert to list of hex pairs
```

```
hex_pairs = hex_sequence.replace("\n", "").split()
```

```
nucleotides = []
```

```
# Map Hex to Nucleotides A, T, G, C based on hex value mod 4
```

```
hex_to_nucleotide = ["A", "T", "G", "C"]
```

```
for pair in hex_pairs:
```

```
    value = int(pair, 16)
```

```
    nucleotides.append(hex_to_nucleotide[value % 4])
```

```
# Convert to string for sequence representation
```

```
nucleotide_sequence = "".join(nucleotides)
```

```
# Step 2: Protein Folding Representation
```

```
# Use angles derived from hex values to simulate folding
```

```
angles = [int(pair, 16) % 360 for pair in hex_pairs]
```

```
# Generate folding diagram (X, Y coordinates)
```

```
x, y = [0], [0]
```

```
for angle in angles:
```

```
    x.append(x[-1] + 1 * (angle % 2))
```

Conversation URL:

<https://chatgpt.com/c/678d4ead-b5d0-8011-a6b6-90a1e4c8650d>

Title:

Prompt:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np

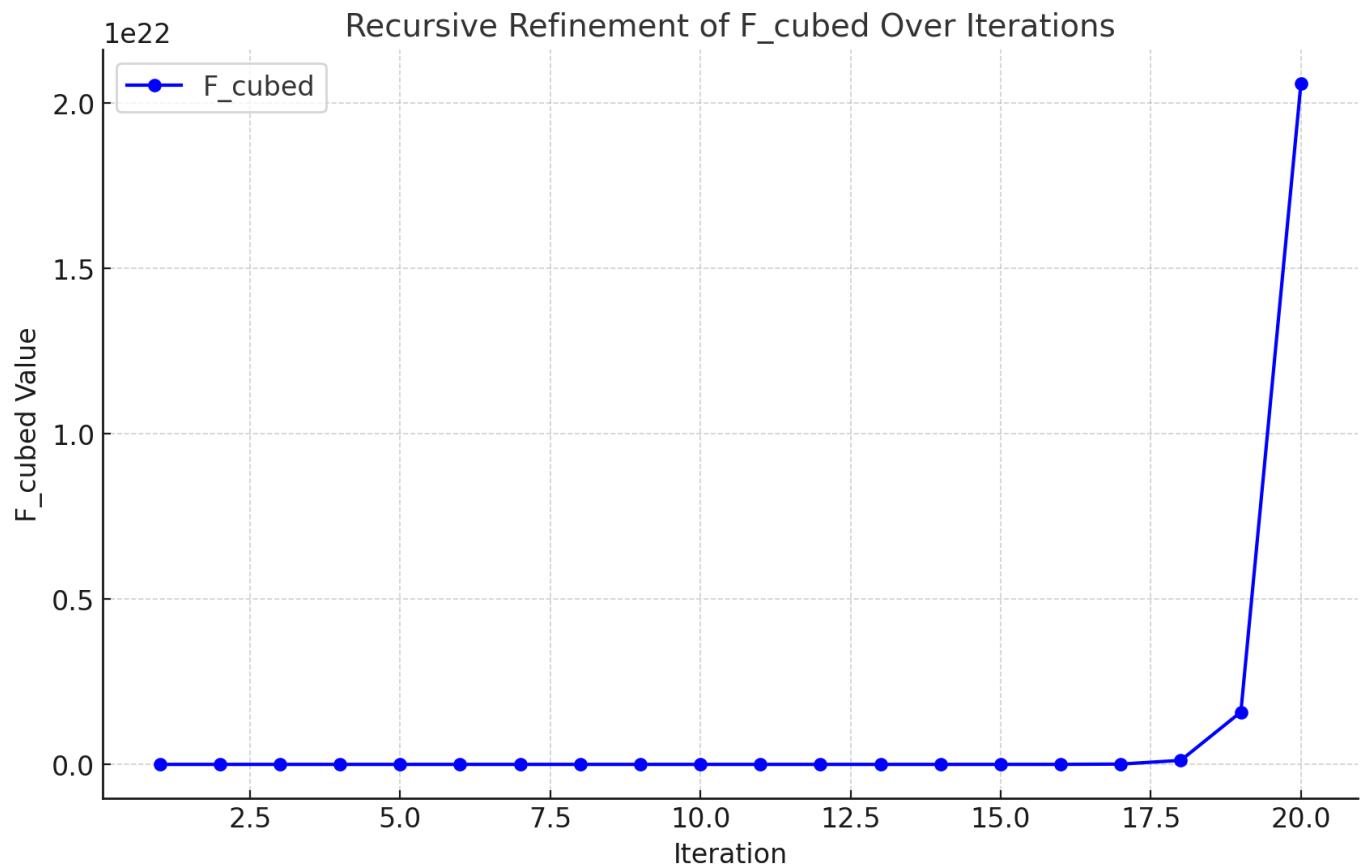
# Generate a simulated 3D fold representation for the peptide
# Hydrophobic residues (A2) and charged residues (B2) are mapped in different regions

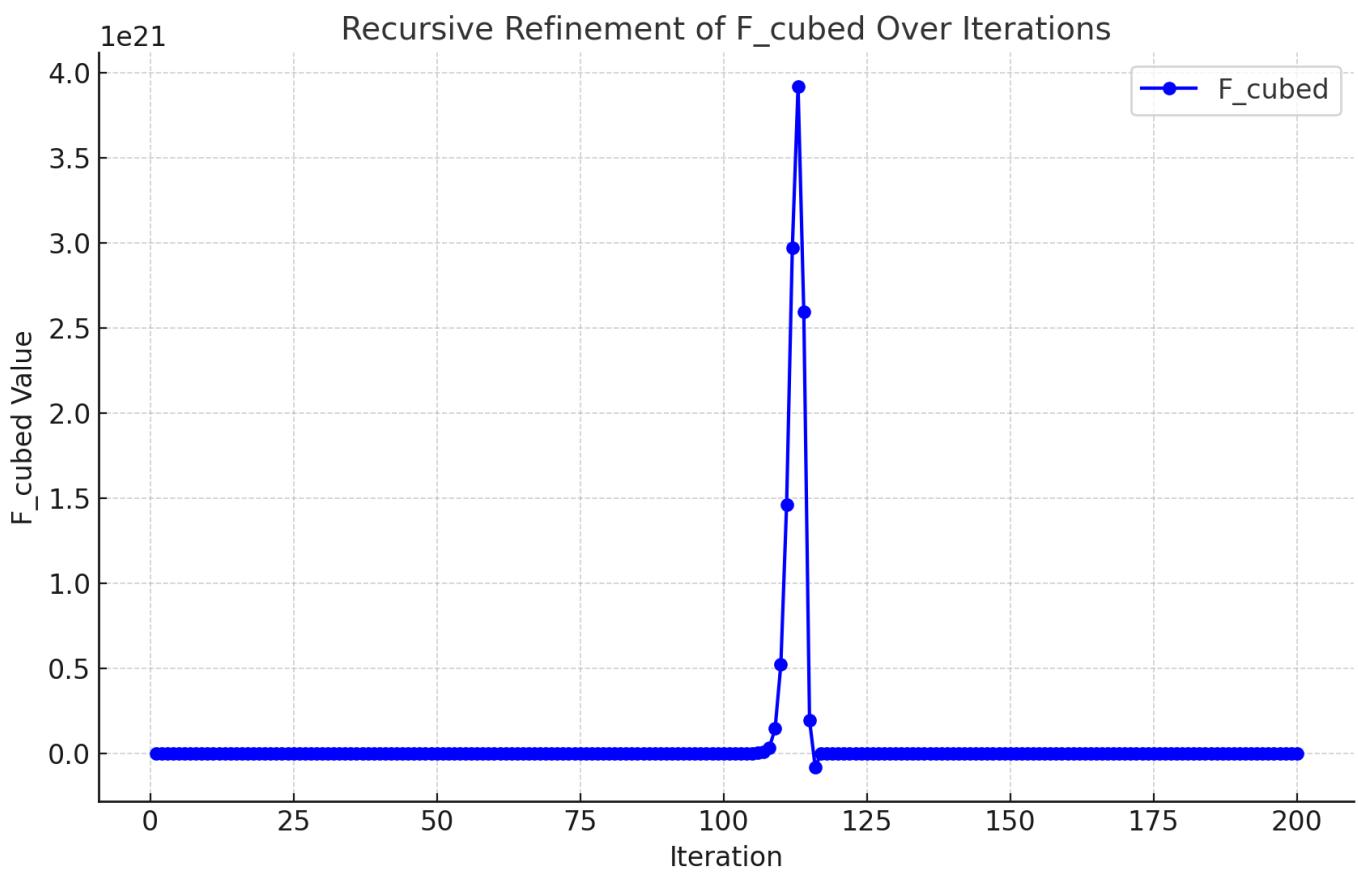
# Simulate a peptide folding path
np.random.seed(42)
x = np.cumsum(np.random.randn(20))
y = np.cumsum(np.random.randn(20))
z = np.cumsum(np.random.randn(20))

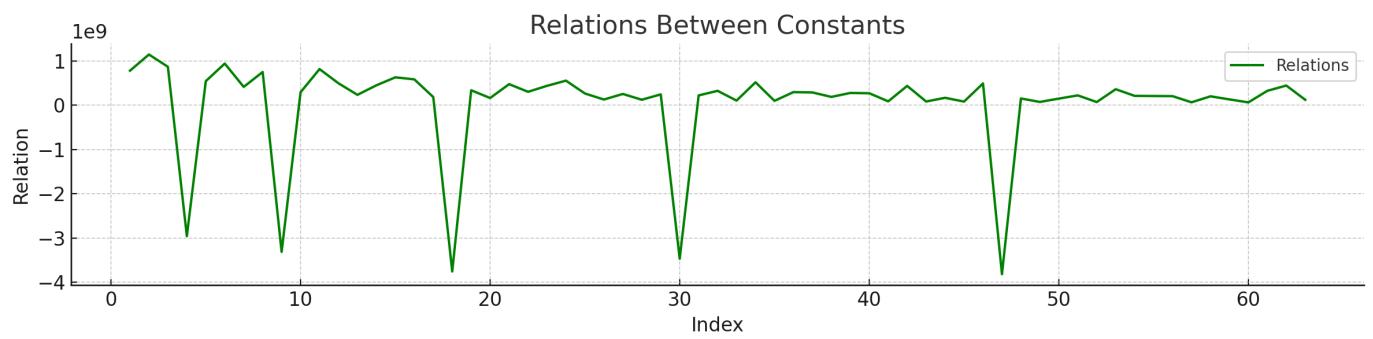
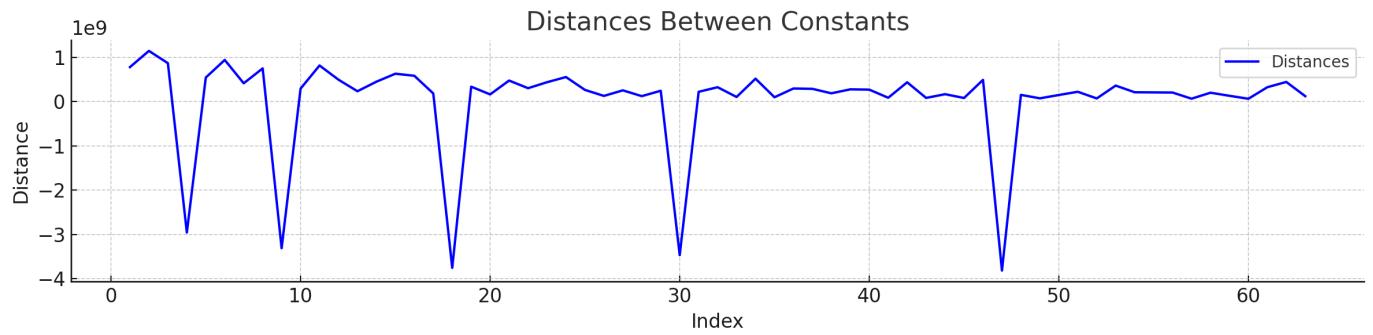
# Define residues as hydrophobic (A2) or charged/polar (B2)
residues = [
    'P', 'Q', 'A', 'W', 'P', 'I', 'A', 'K', 'K', 'K', 'K',
    'S', 'C', 'E', 'C', 'G', 'D', 'R', 'R', 'A'
]
colors = ['red' if r in ['K', 'R', 'E', 'C', 'S', 'D'] else 'blue' for r in residues]

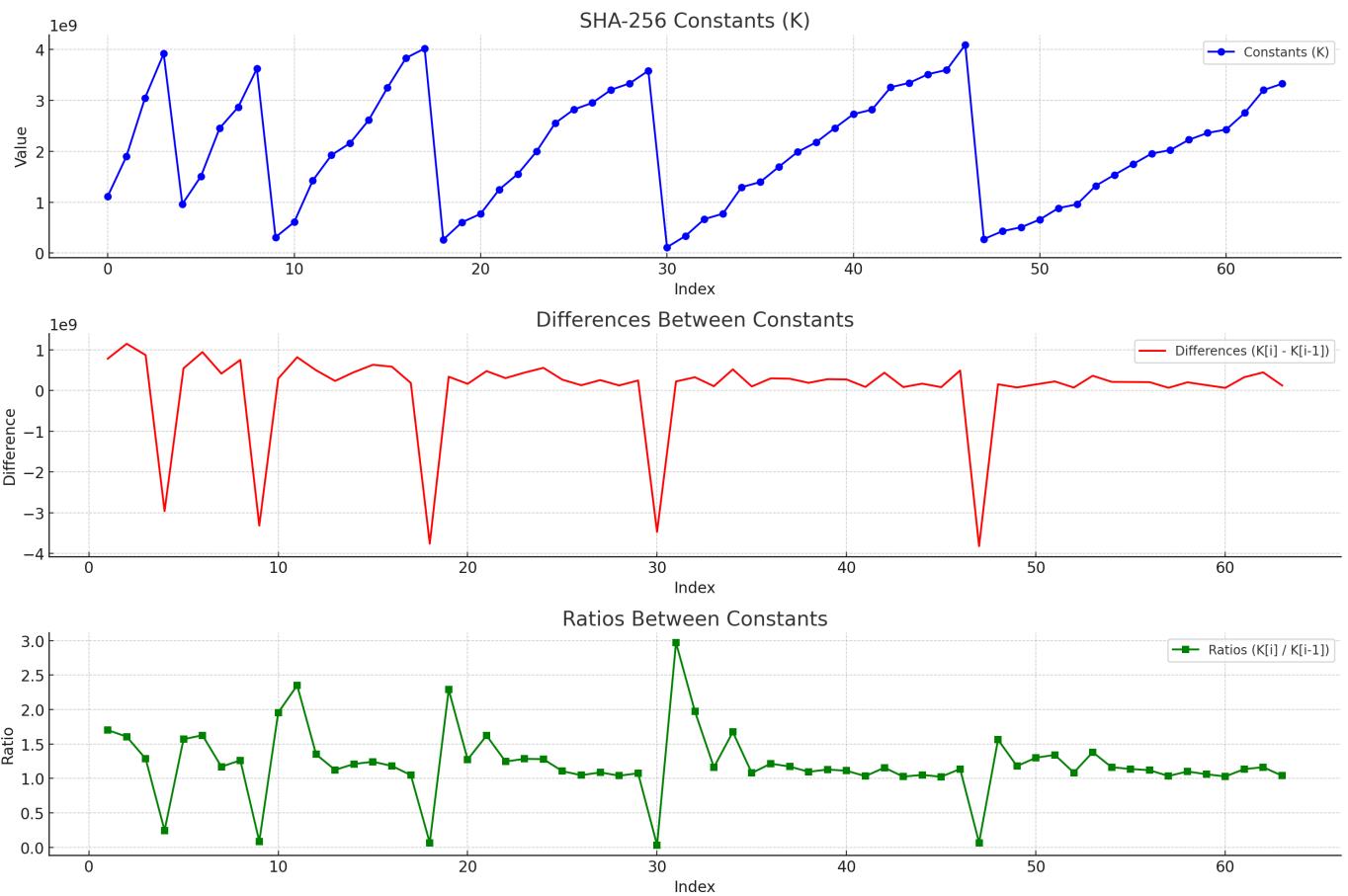
# Plot the folding representation
fig = plt.figure(figsize=(12, 6))
ax = fig.add_subplot(111, projection='3d')
scatter = ax.scatter(x, y, z, c=colors, s=100, alpha=0.8)
ax.plot(x, y, z, color='gray', alpha=0.5)
for i, txt in enumerate(residues):
    ax.text(x[i], y[i], z[i], txt, fontsize=9)

ax.set_title("3D Representation of Peptide Folding")
```

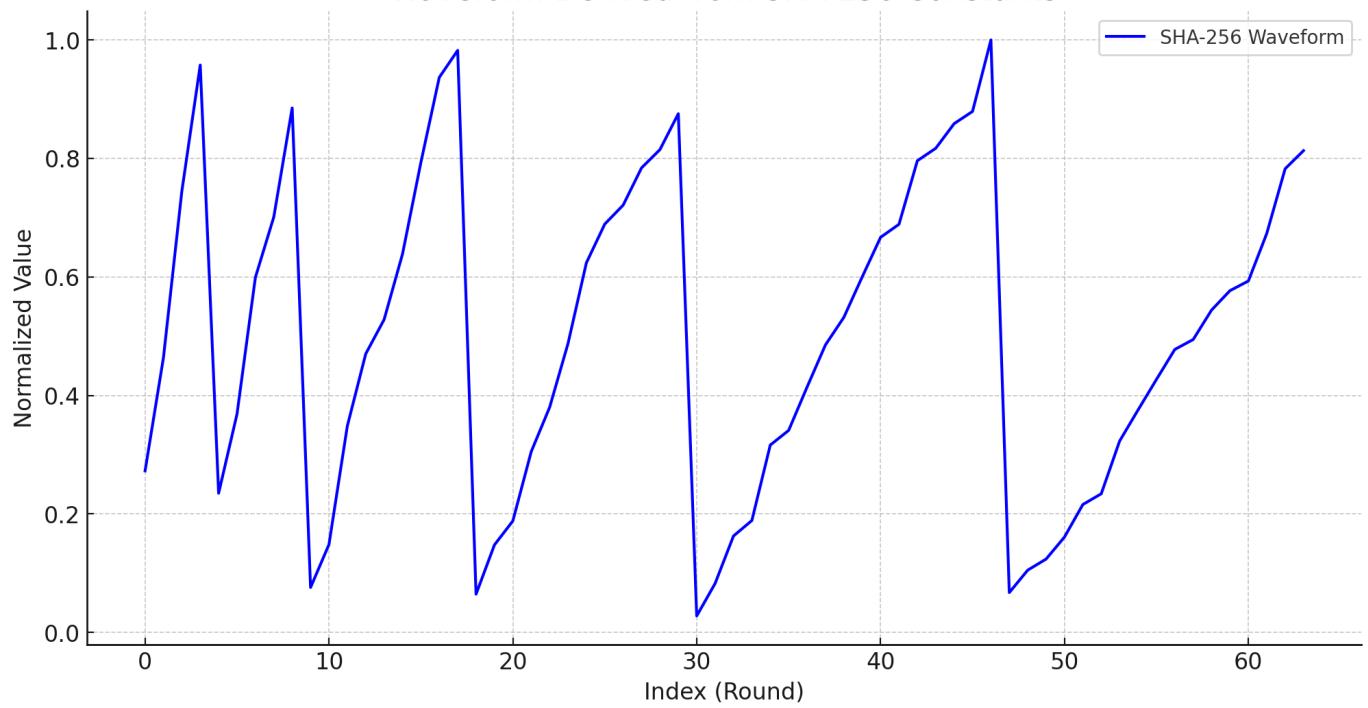




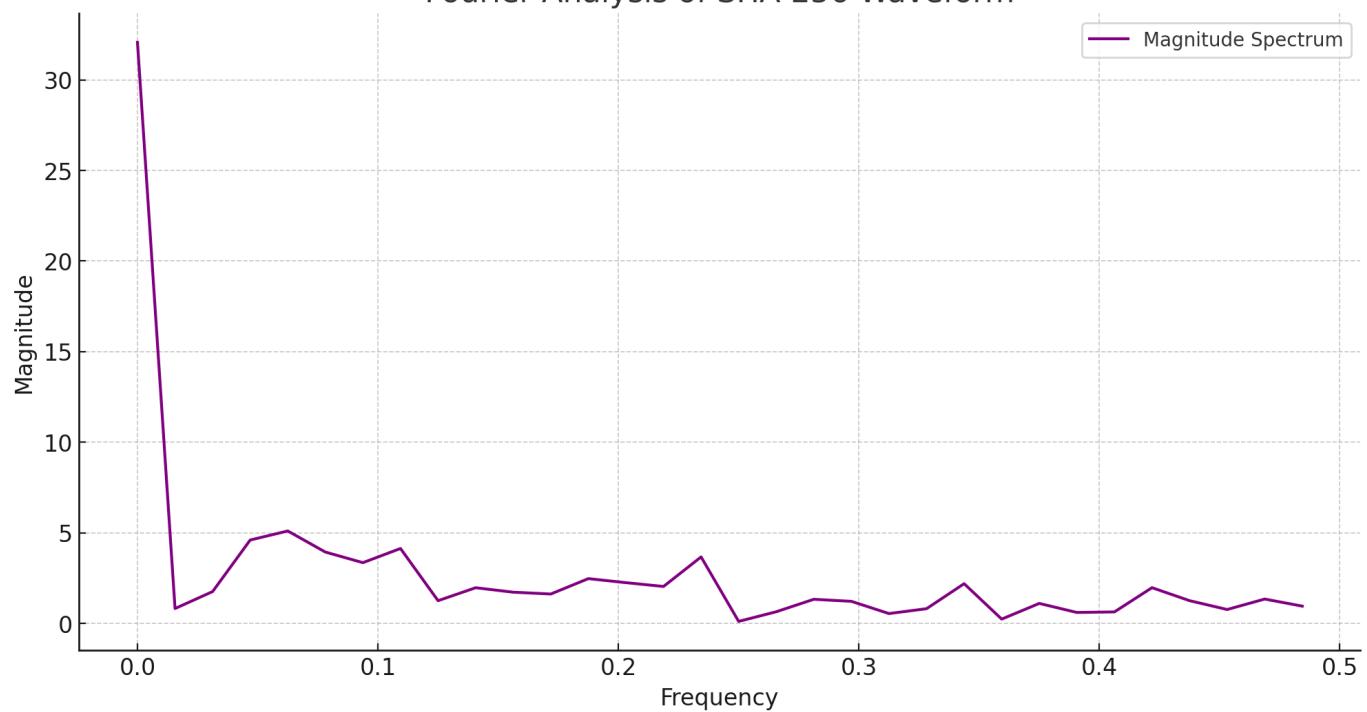




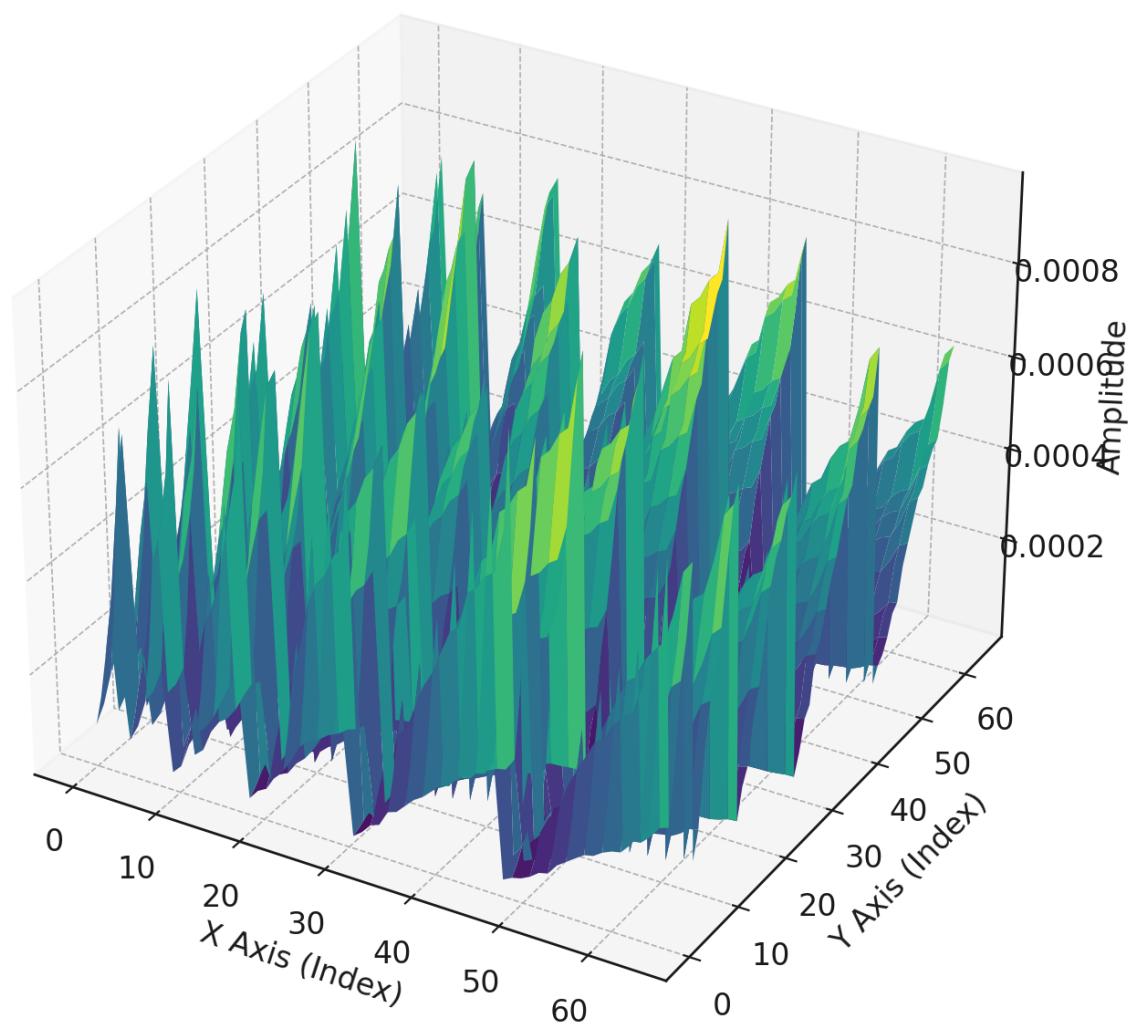
Waveform Derived from SHA-256 Constants



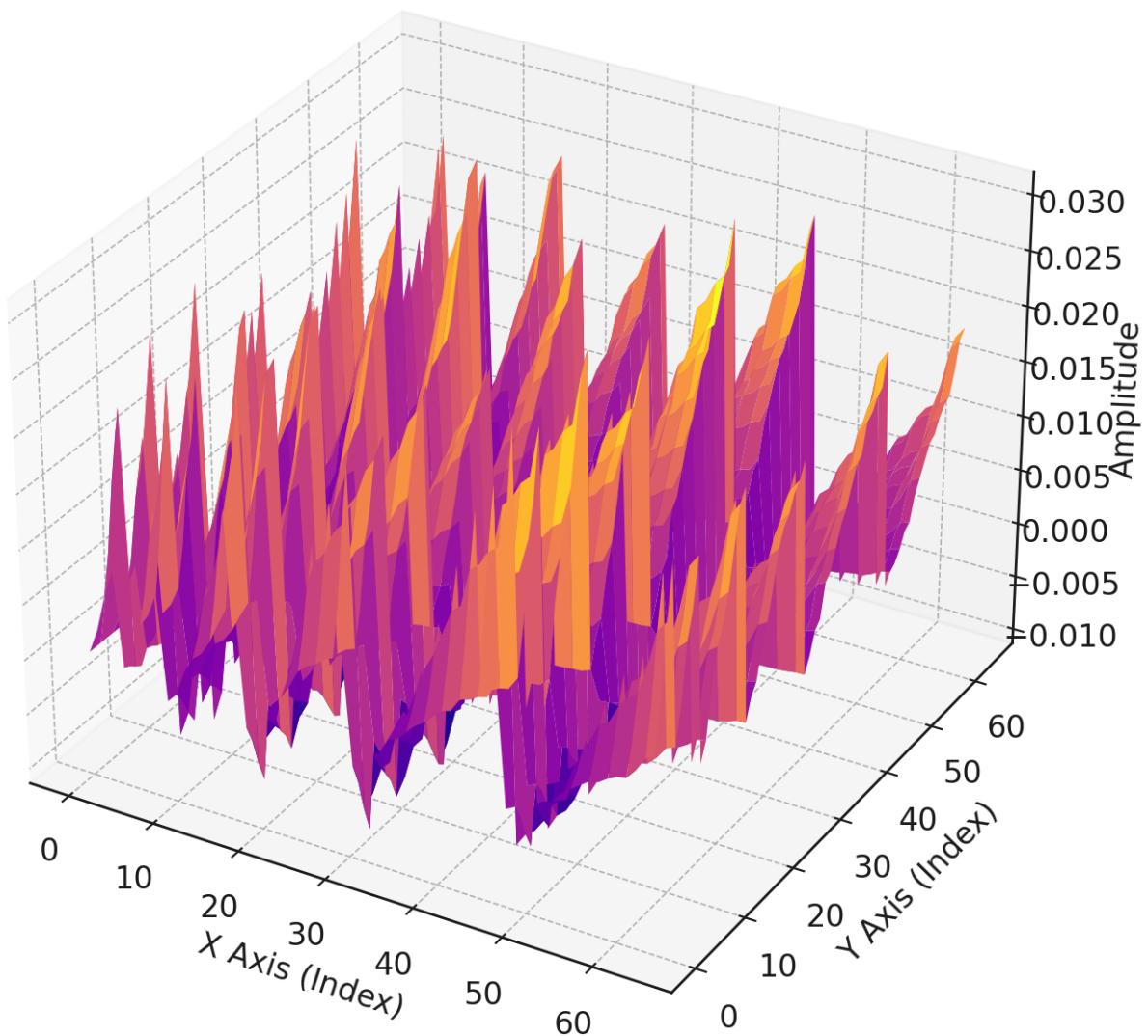
Fourier Analysis of SHA-256 Waveform



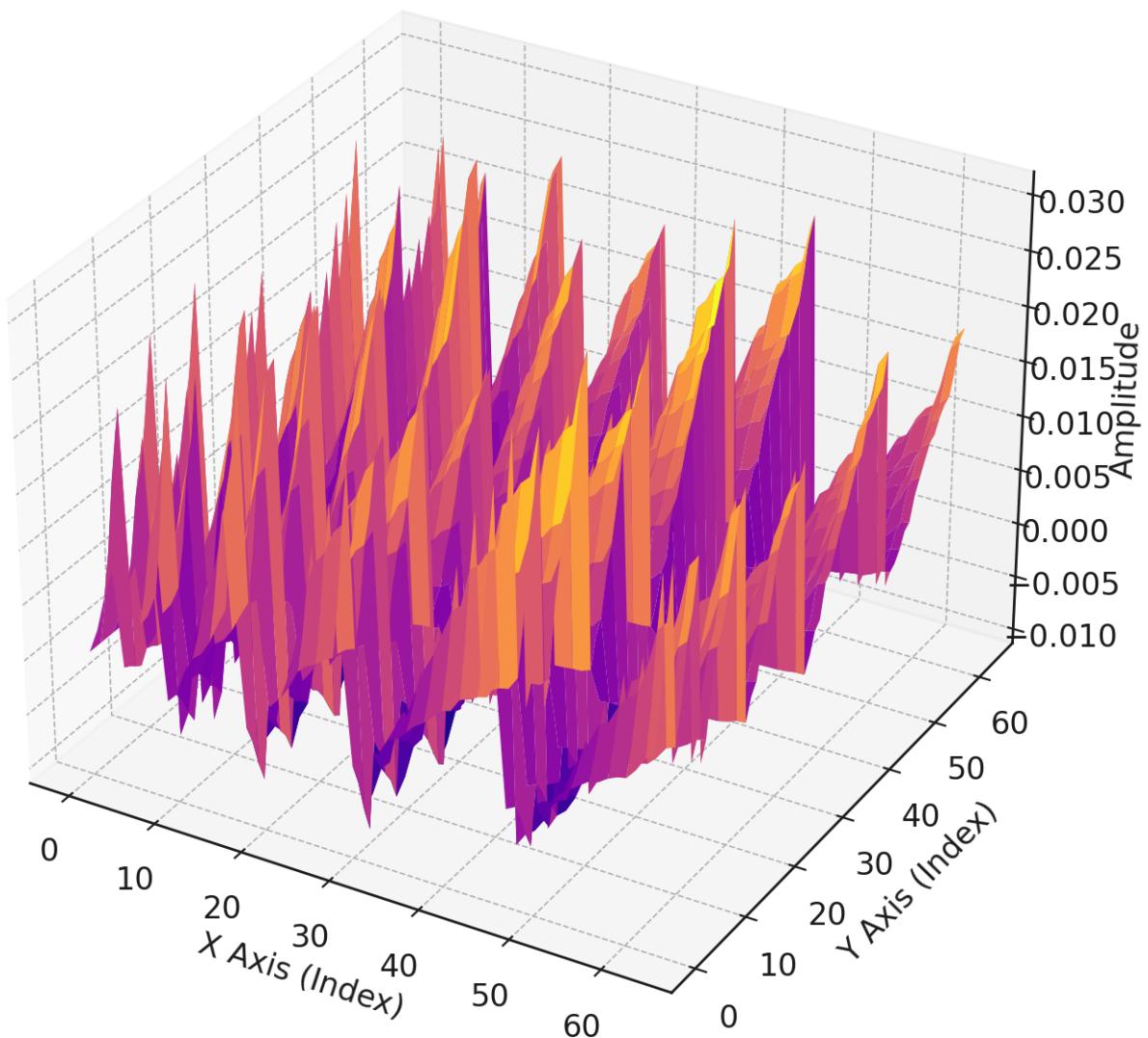
3D Quantum-Inspired Waveform Derived from SHA-256 Constants

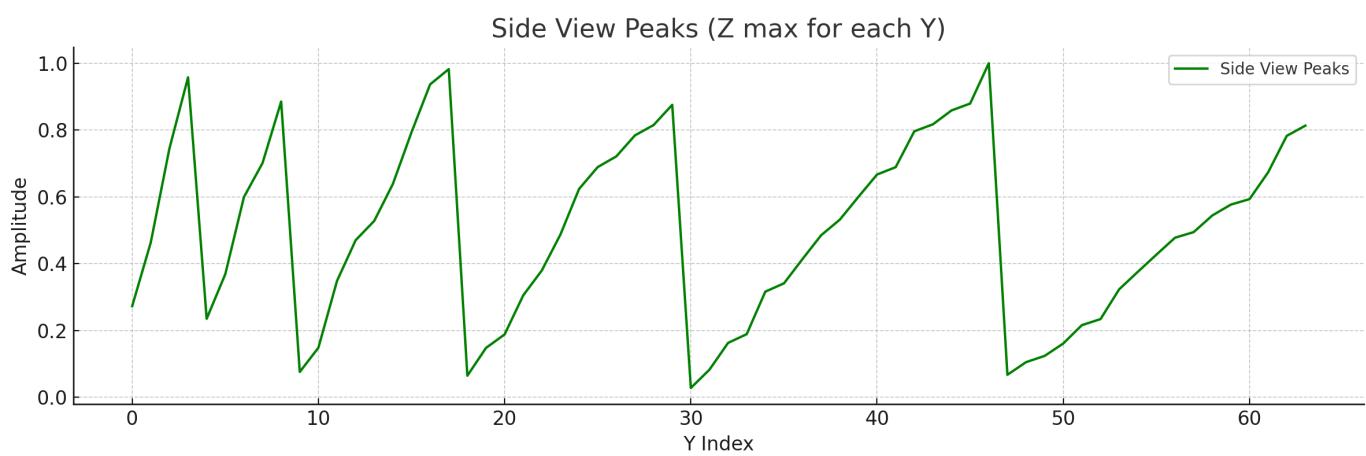
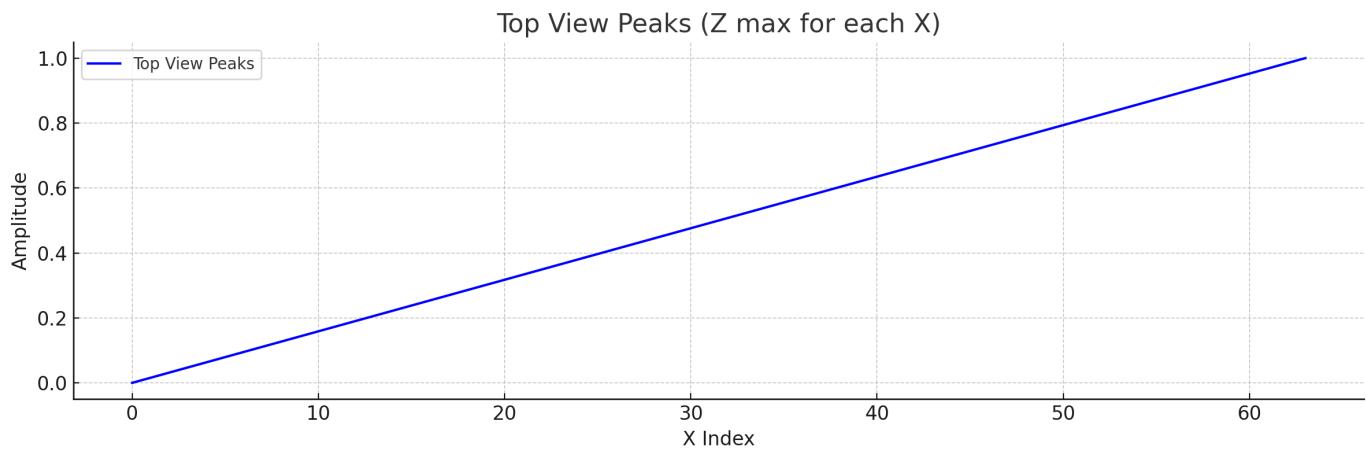


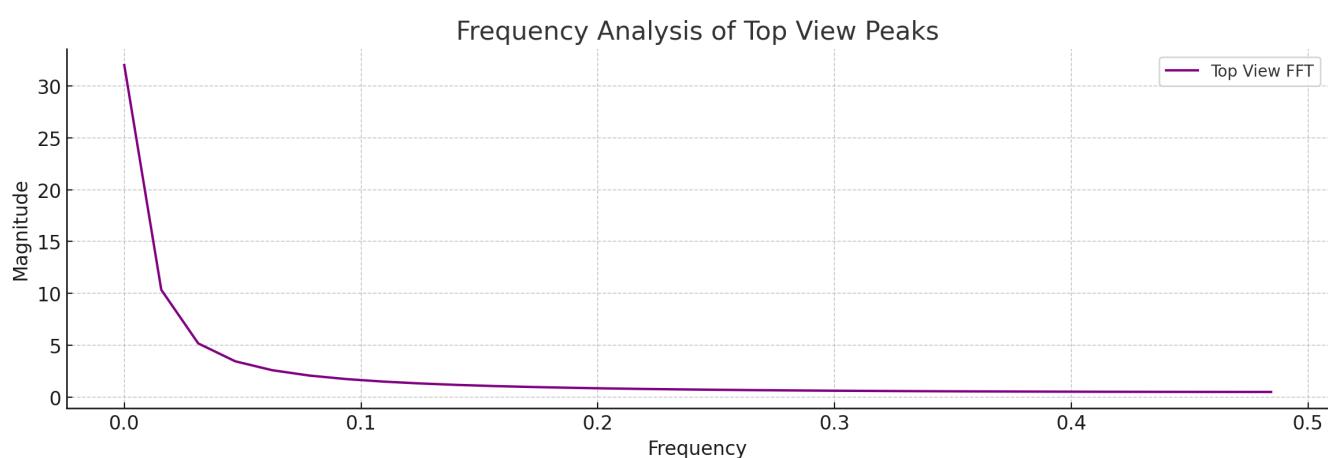
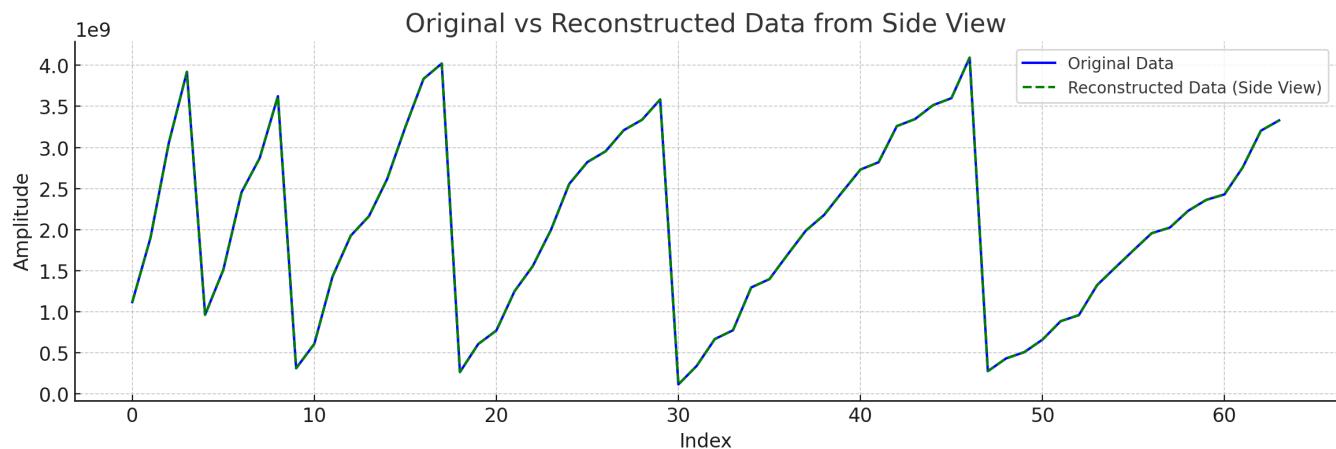
Refined 3D Waveform with Interaction (Kinetics and Quantum)



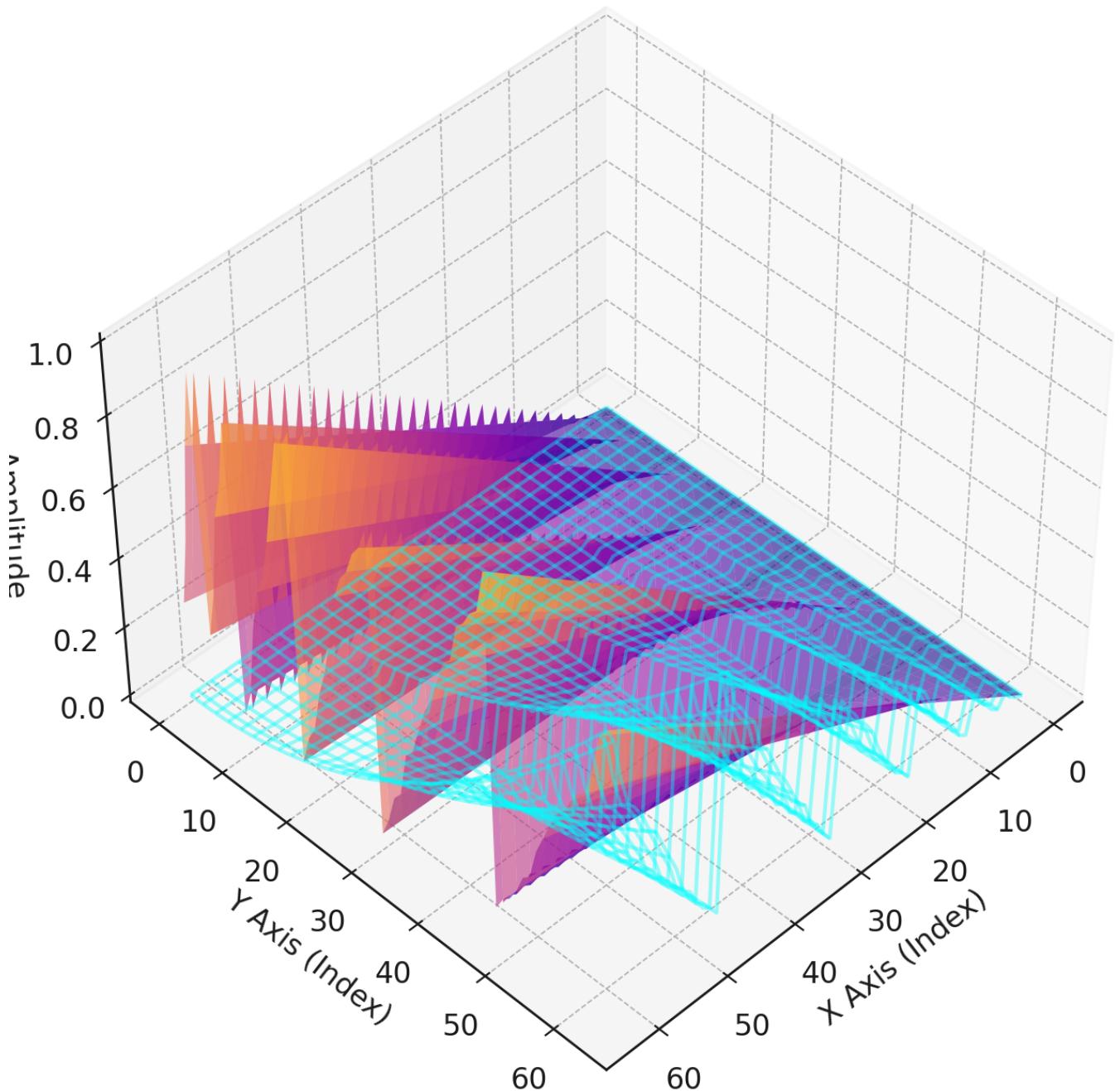
Refined 3D Waveform with Interaction (Kinetics and Quantum)



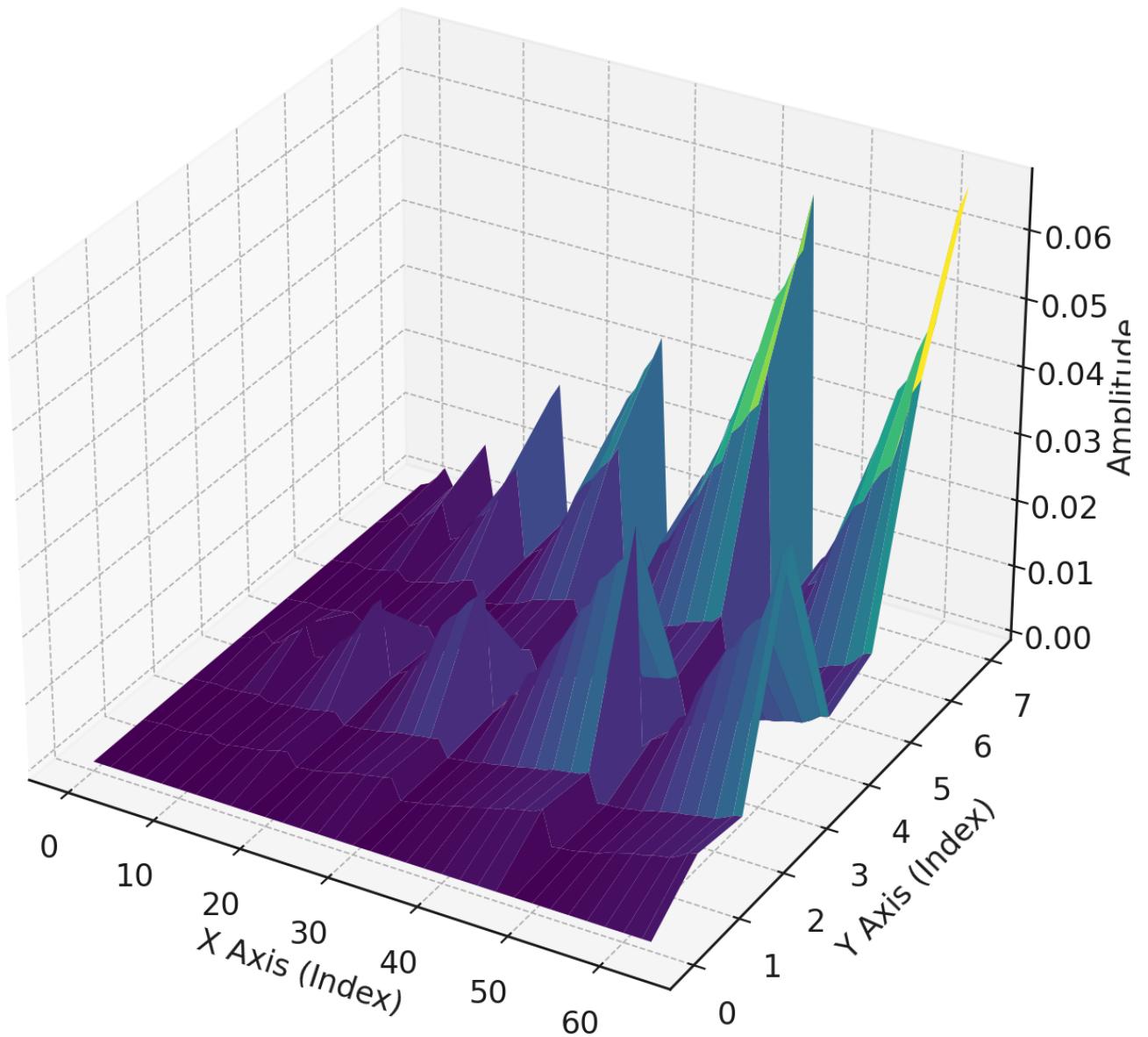




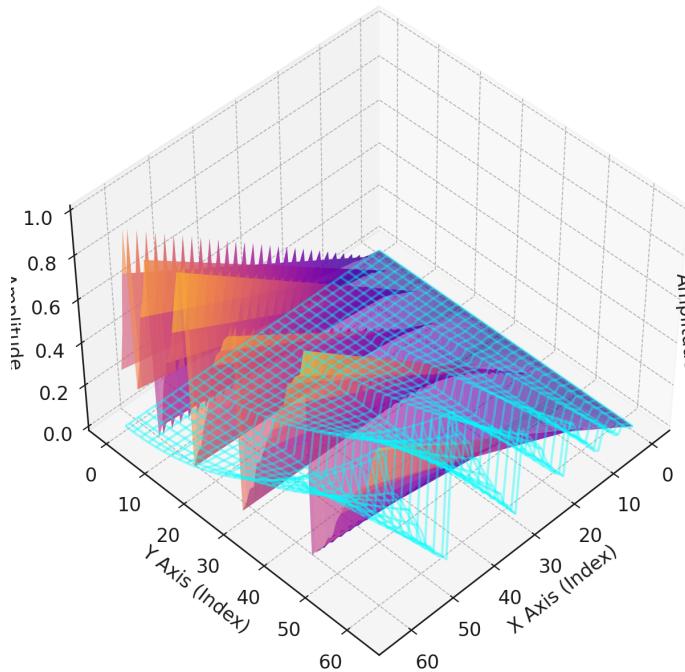
Diagonal View with Extended Interactions



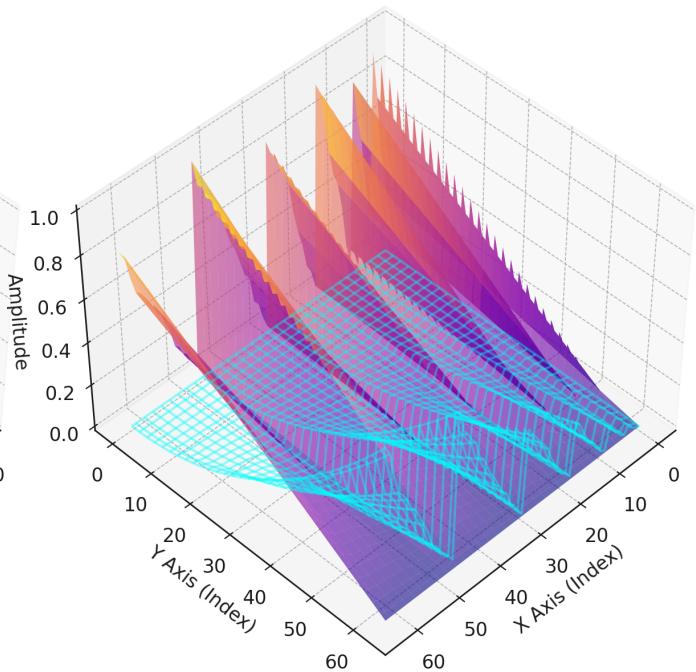
Corrected Hash-Augmented 3D Waveform



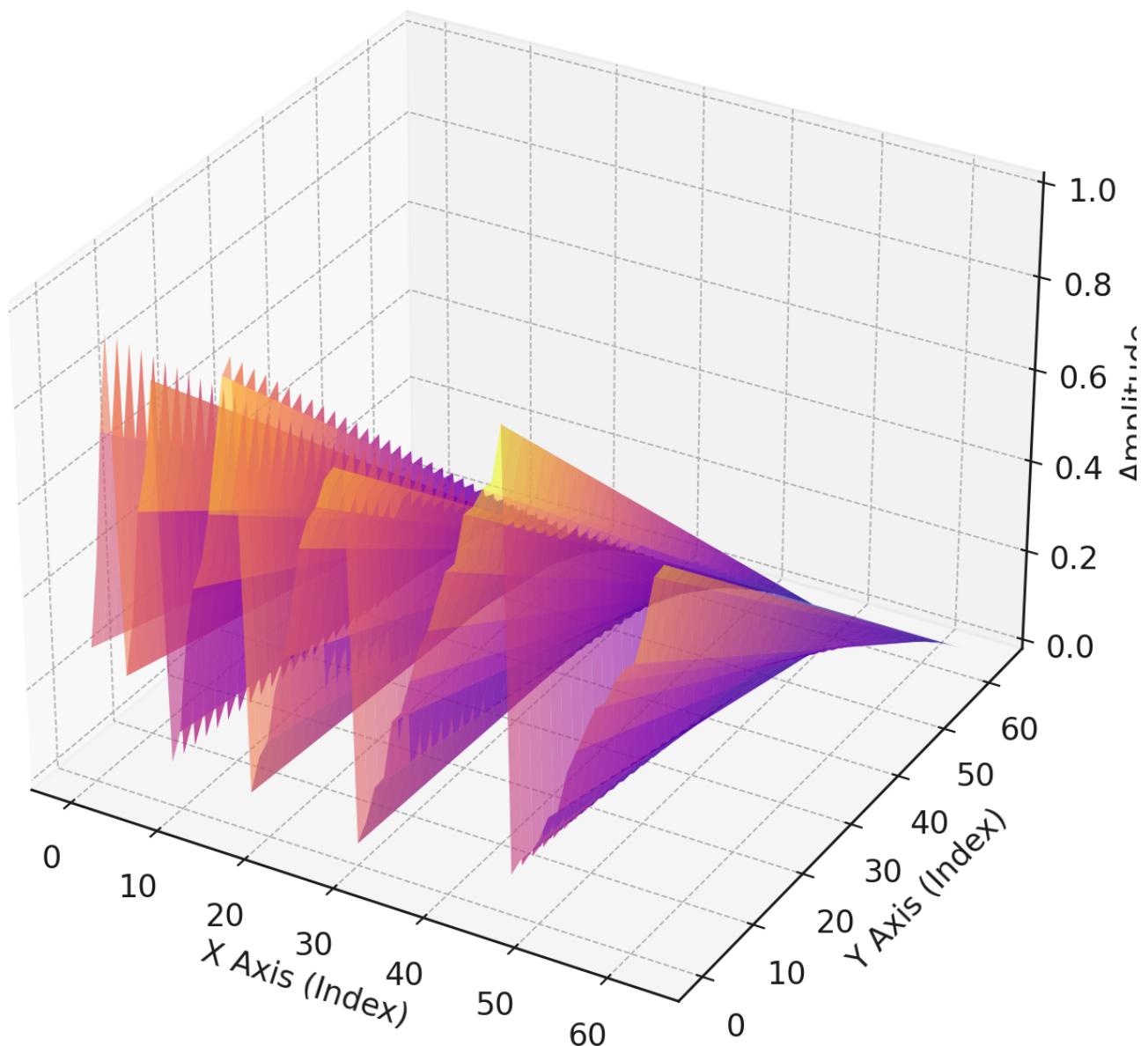
Original Diagonal View with Extended Interactions



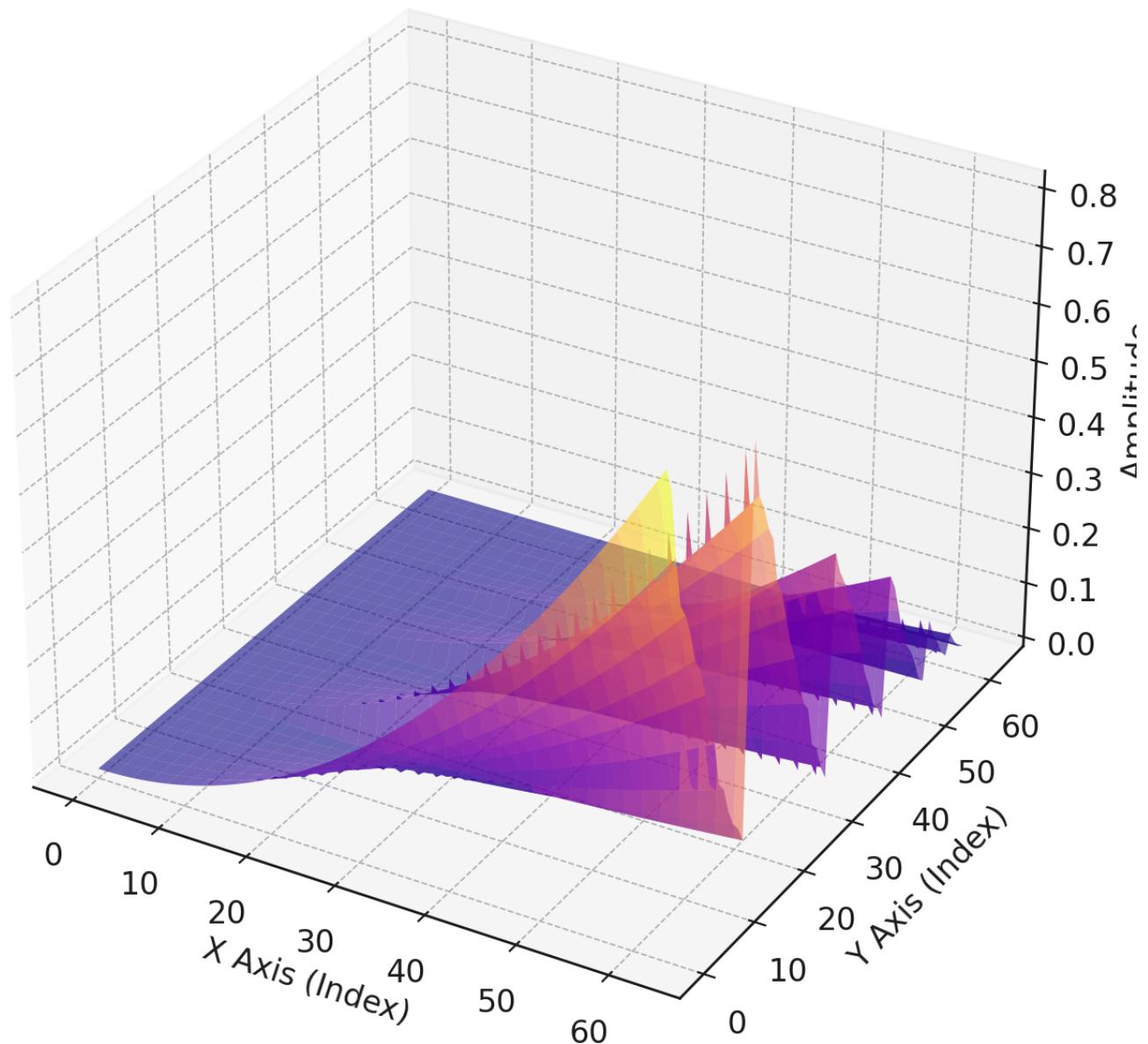
Rotated Grid (90° Counterclockwise)



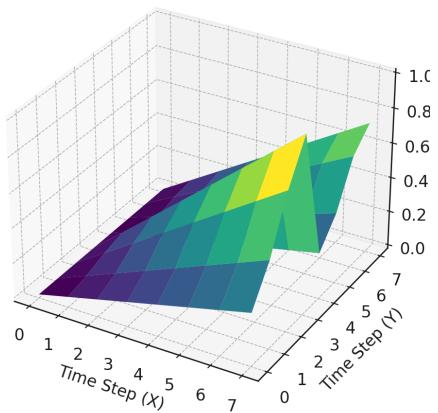
Rotated Grid - Blue Grid Rotated 90 Degrees



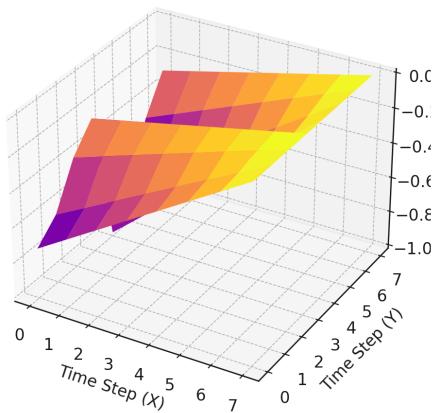
Side View of Rotated Grid



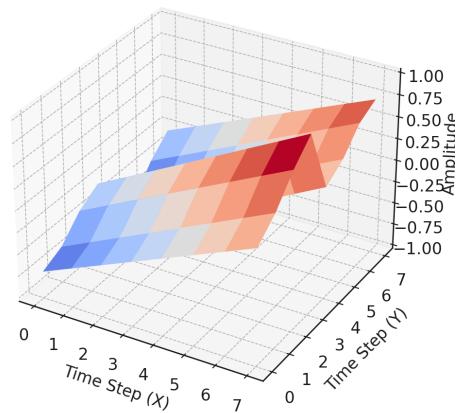
Forward Kinetics Lattice



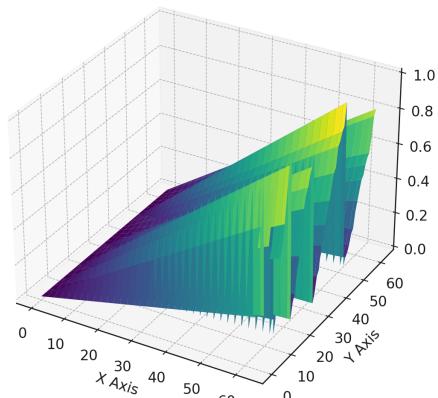
Mirrored Kinetics Lattice



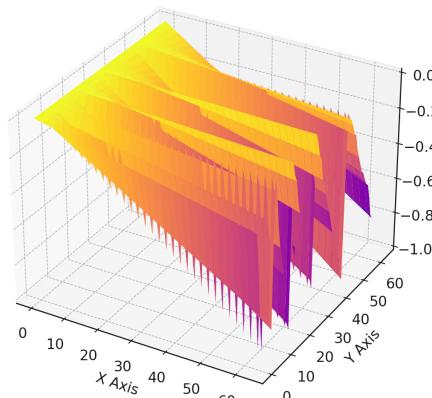
Refined Interaction Lattice



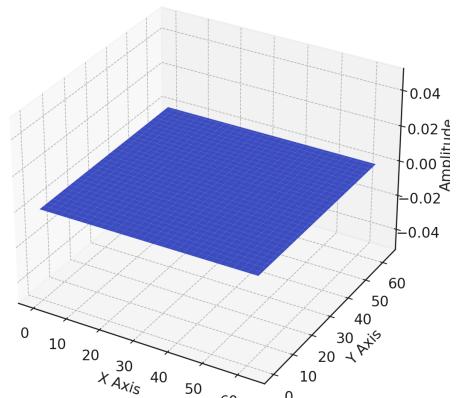
Forward Kinetics Lattice



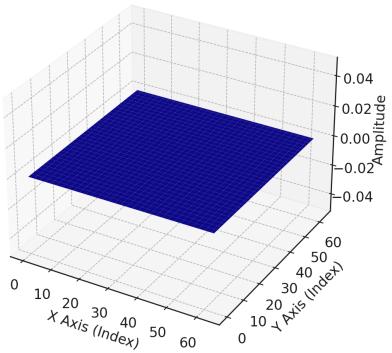
Mirrored Kinetics Lattice



Refined Interaction Lattice

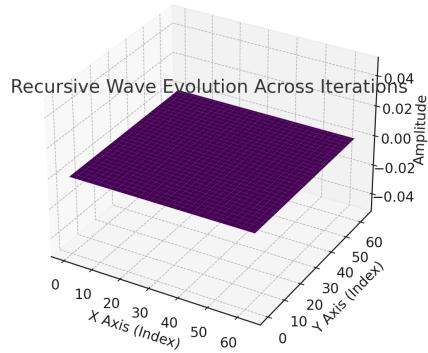


Iteration 1: Forward Lattice



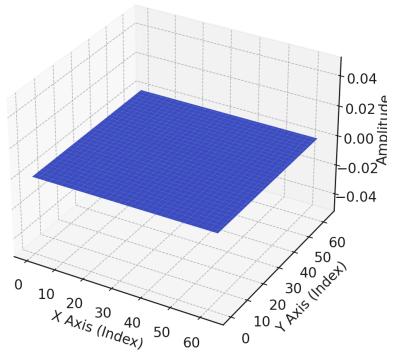
Iteration 2: Forward Lattice

Iteration 1: Mirrored Lattice



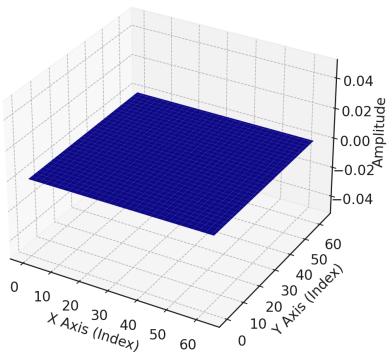
Iteration 2: Mirrored Lattice

Iteration 1: Combined Lattice



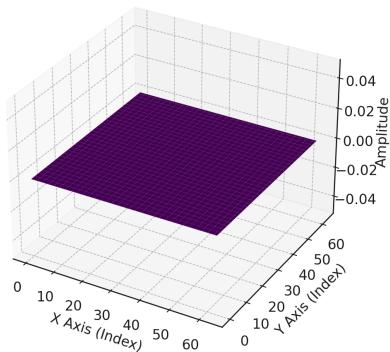
Iteration 2: Combined Lattice

Iteration 3: Forward Lattice



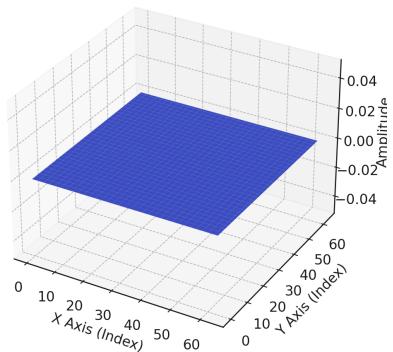
Iteration 4: Forward Lattice

Iteration 3: Mirrored Lattice



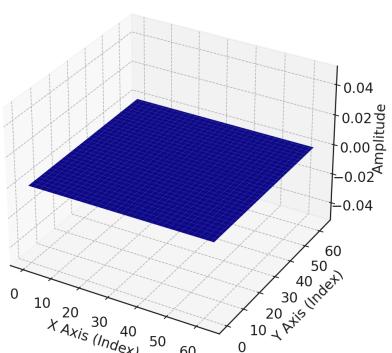
Iteration 4: Mirrored Lattice

Iteration 3: Combined Lattice



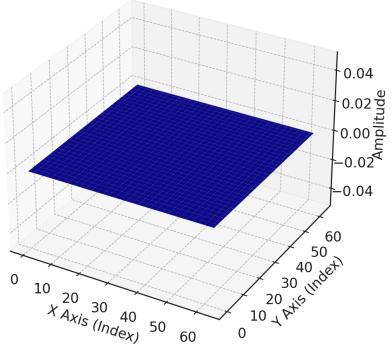
Iteration 4: Combined Lattice

Iteration 5: Forward Lattice

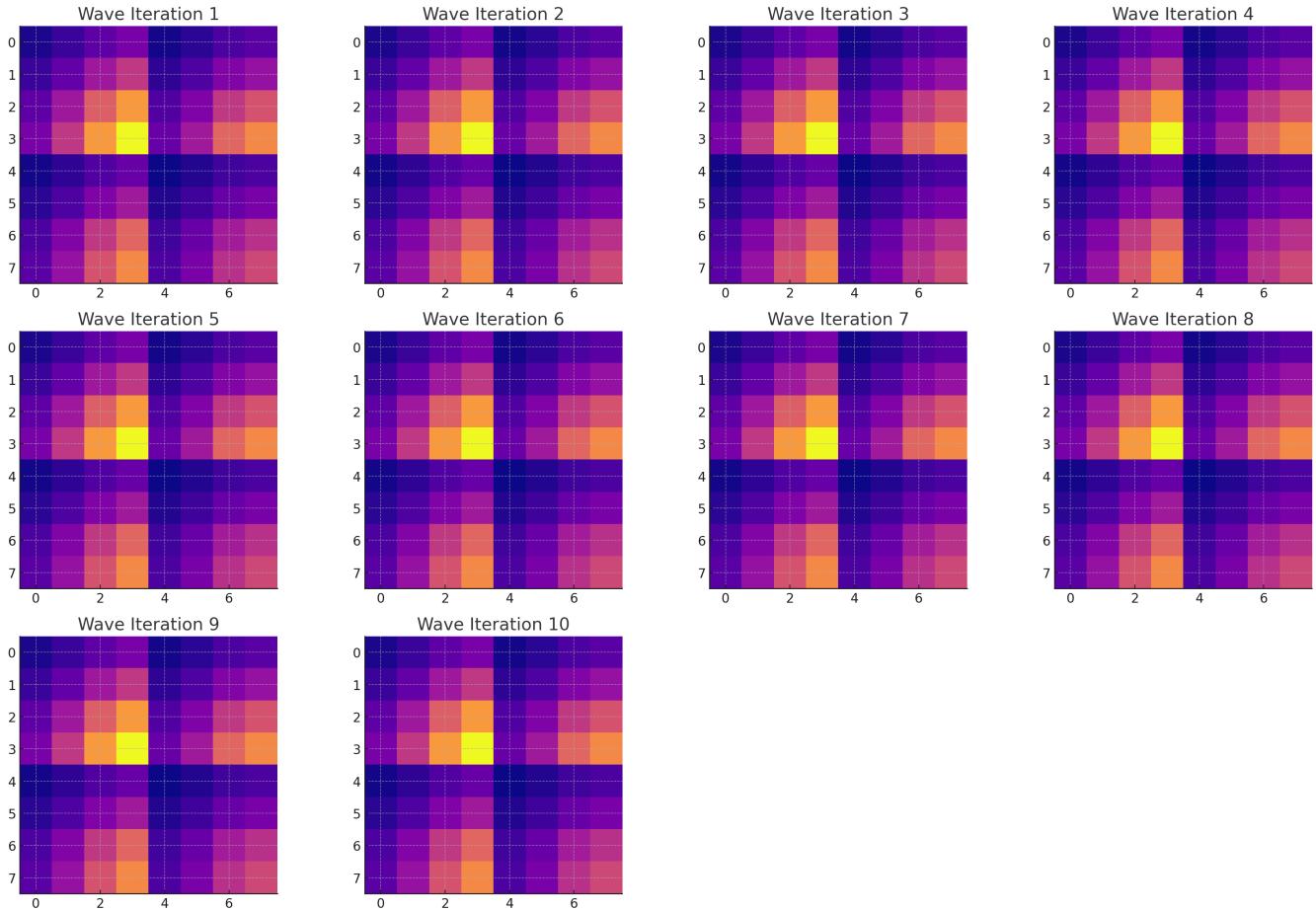


Iteration 5: Mirrored Lattice

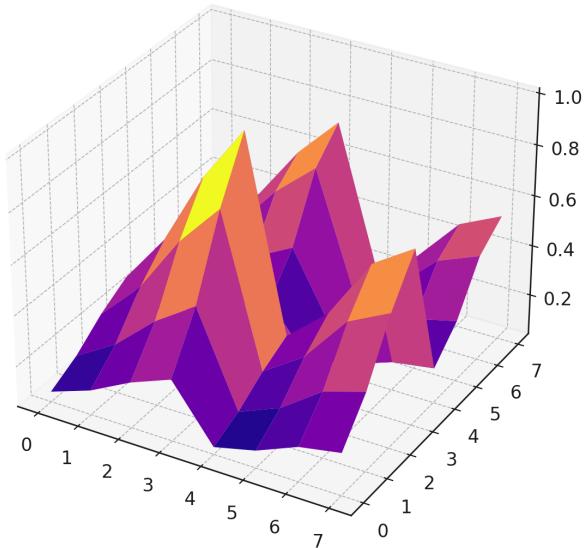
Iteration 5: Combined Lattice



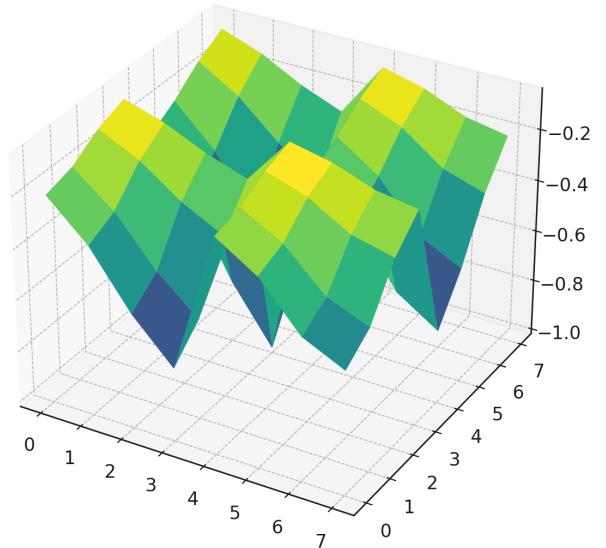
A 3D surface plot representing a function where the output is zero across the entire domain. The horizontal axes are labeled "X Axis (Index)" and "Y Axis (Index)", both ranging from 0 to 60. The vertical axis is labeled "Amplitude", ranging from -0.04 to 0.04. The surface is a perfectly flat plane at the zero level.



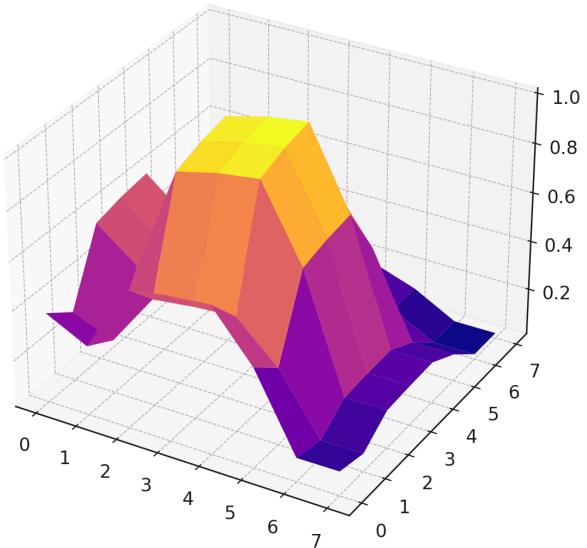
Original Refined Lattice



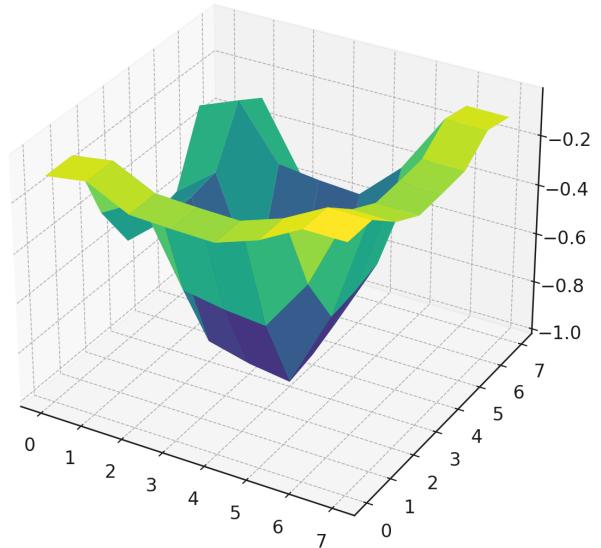
Quantum Mirrored Lattice

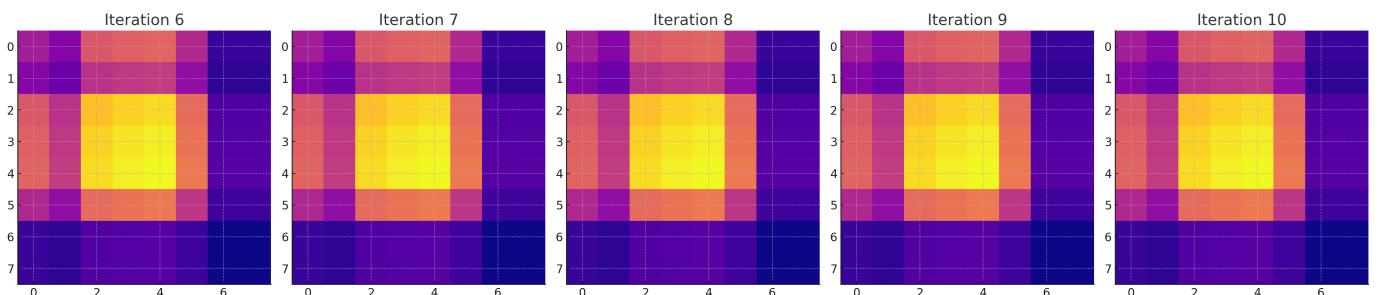
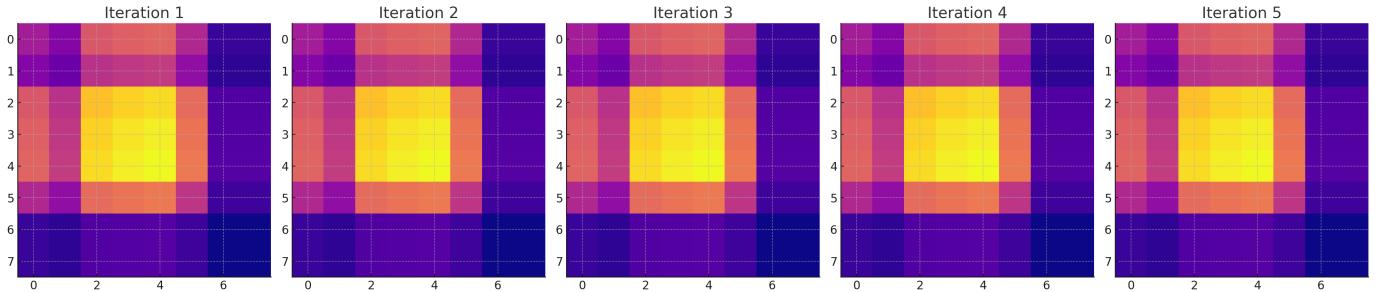


Refined Hash Lattice

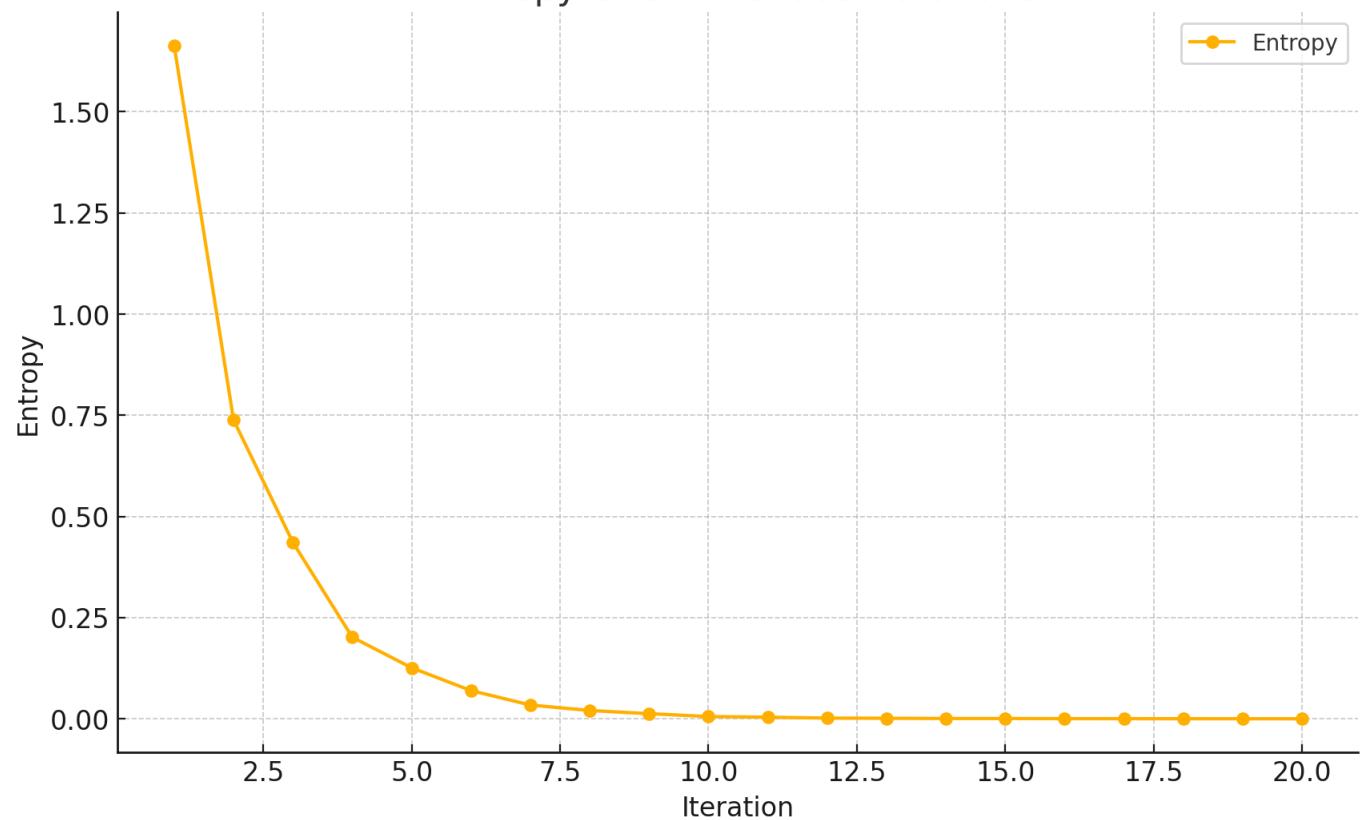


Quantum-Mirrored Hash Lattice

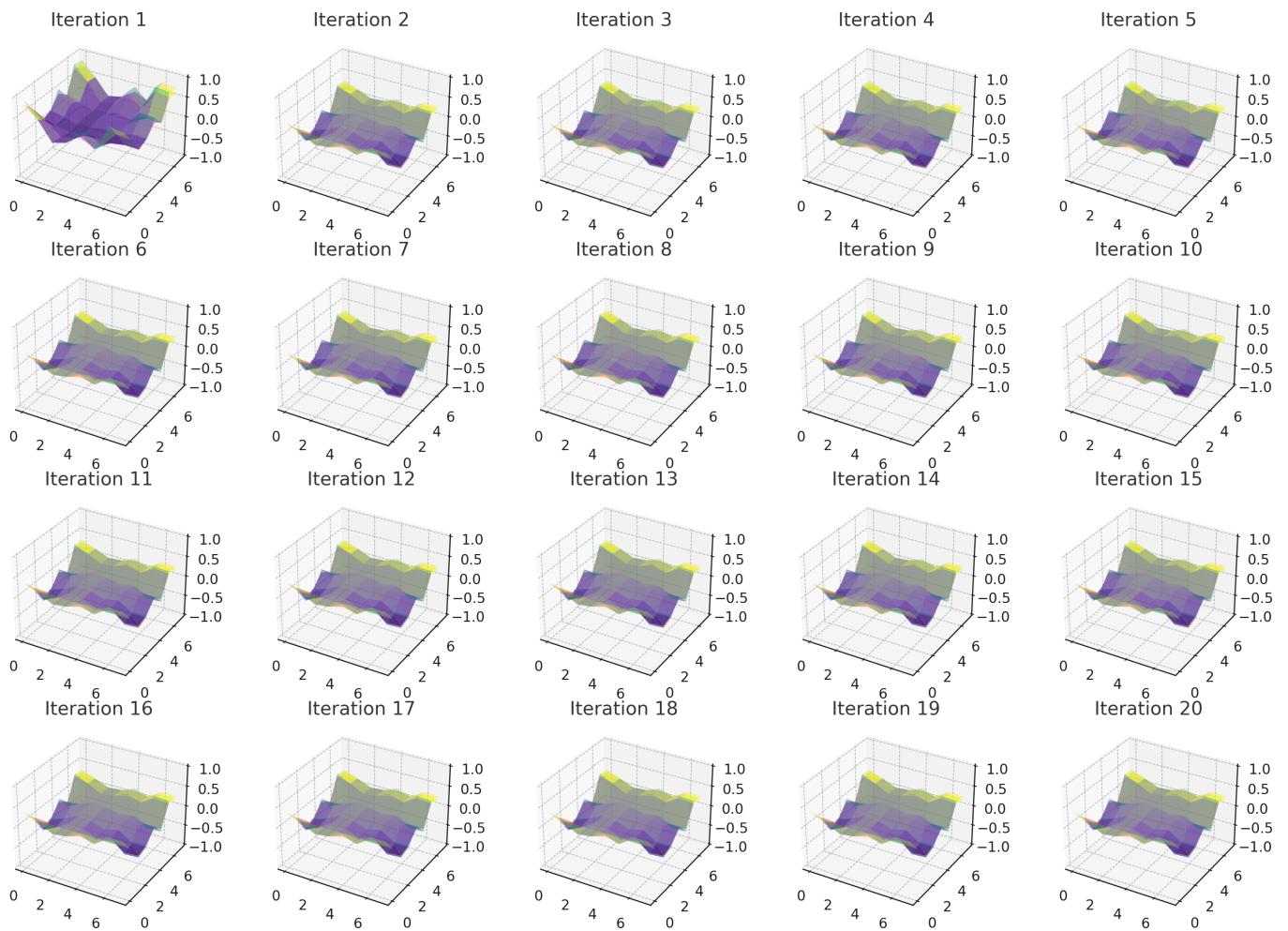


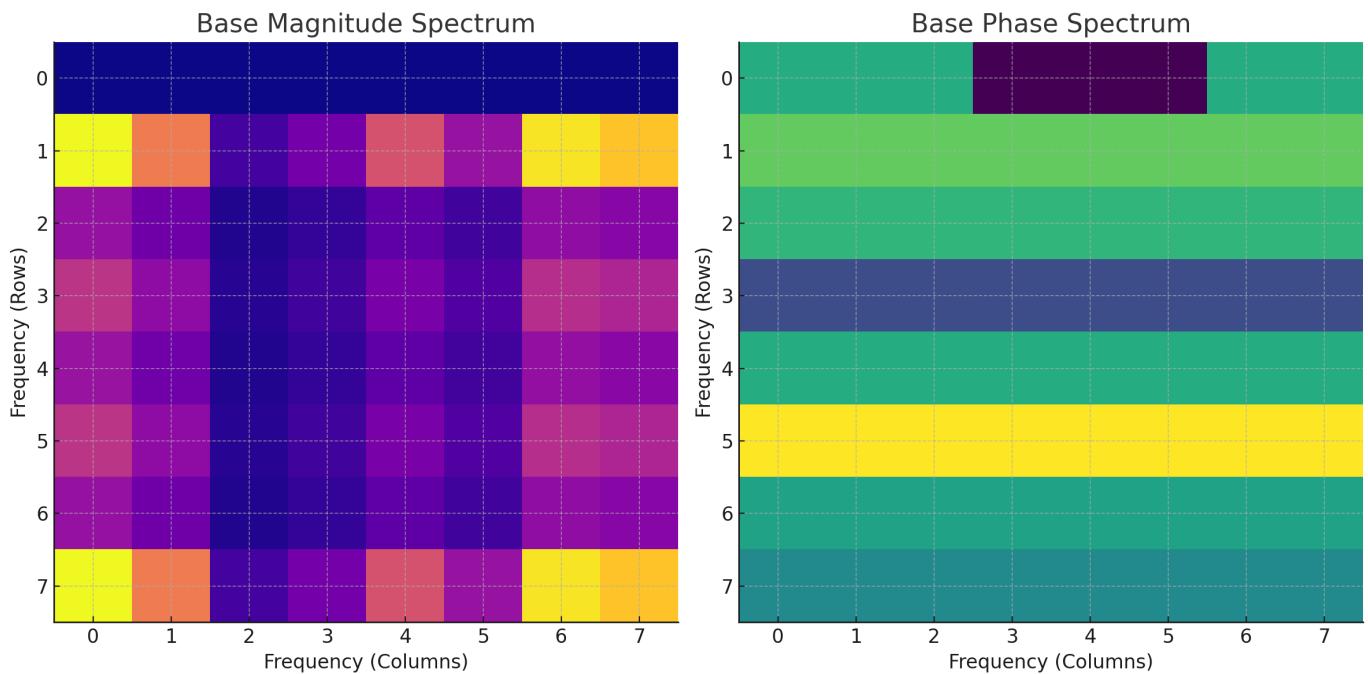


Entropy Over Extended Iterations

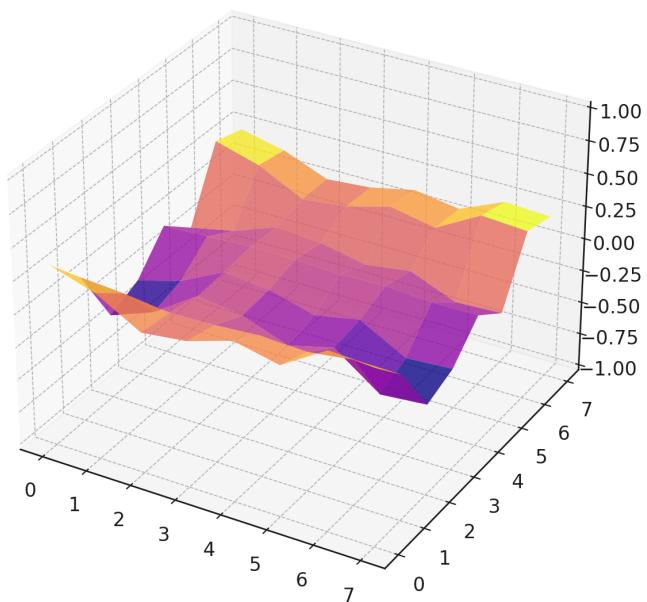


Base vs Perturbed Wave Morphing Over Iterations

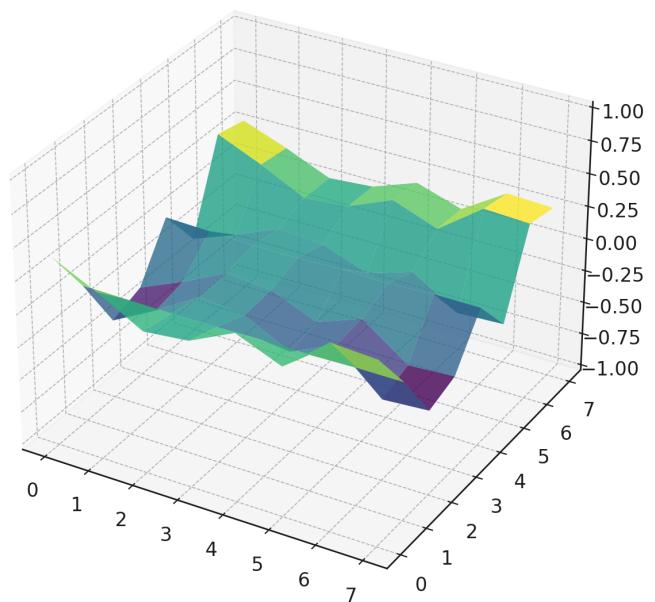




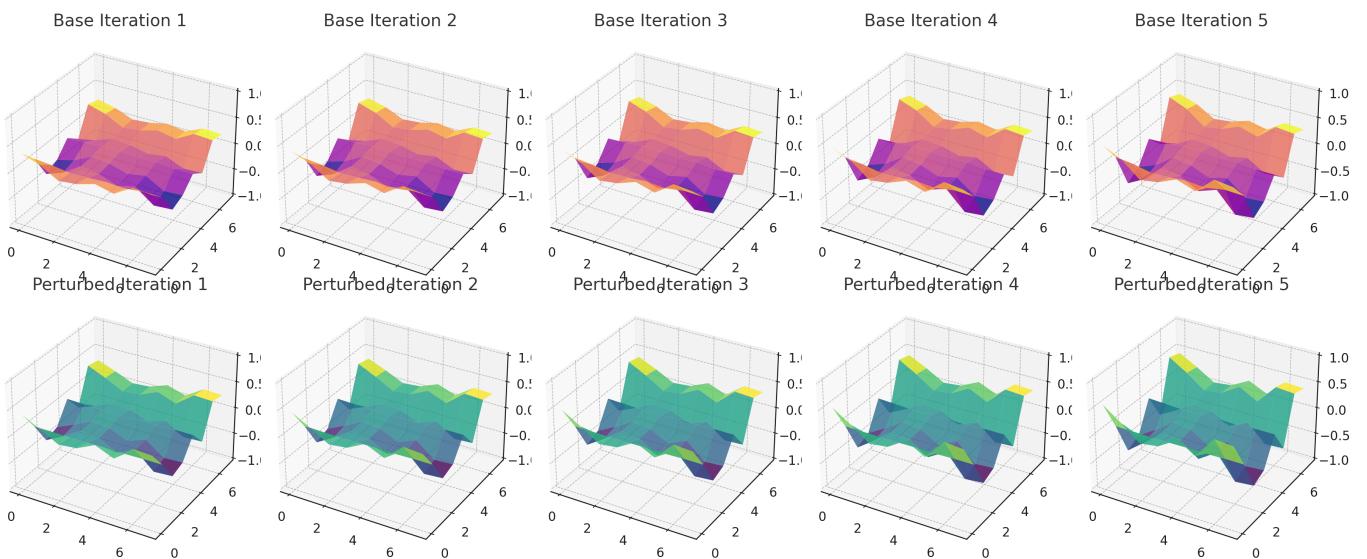
Final Base Waveform



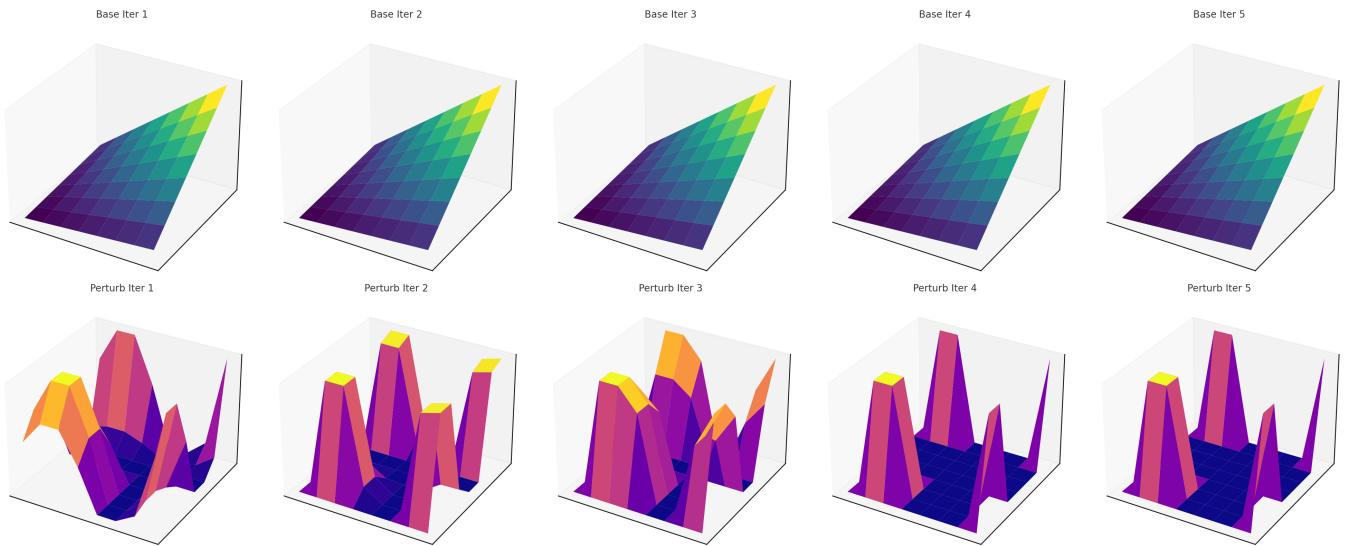
Final Perturbed Waveform

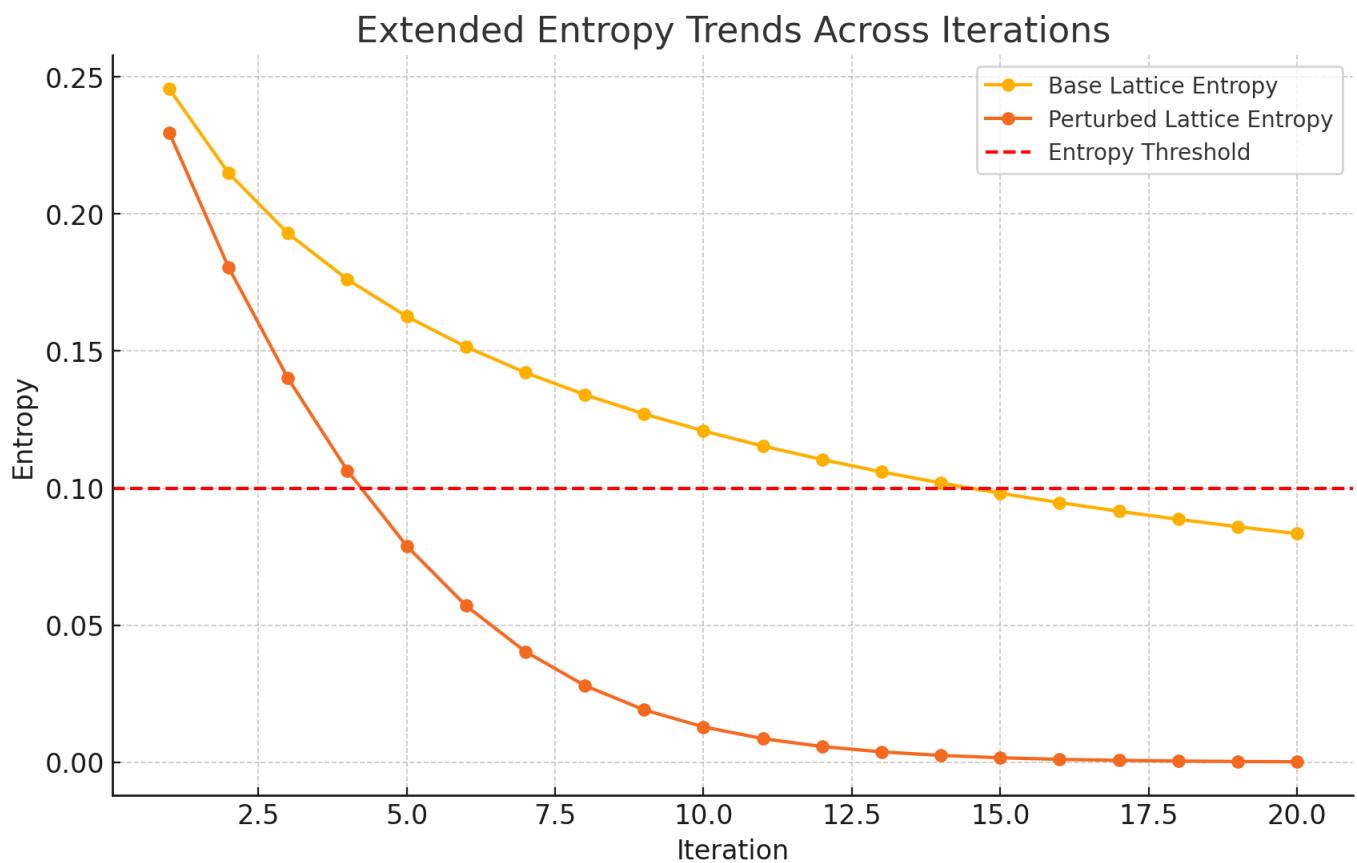


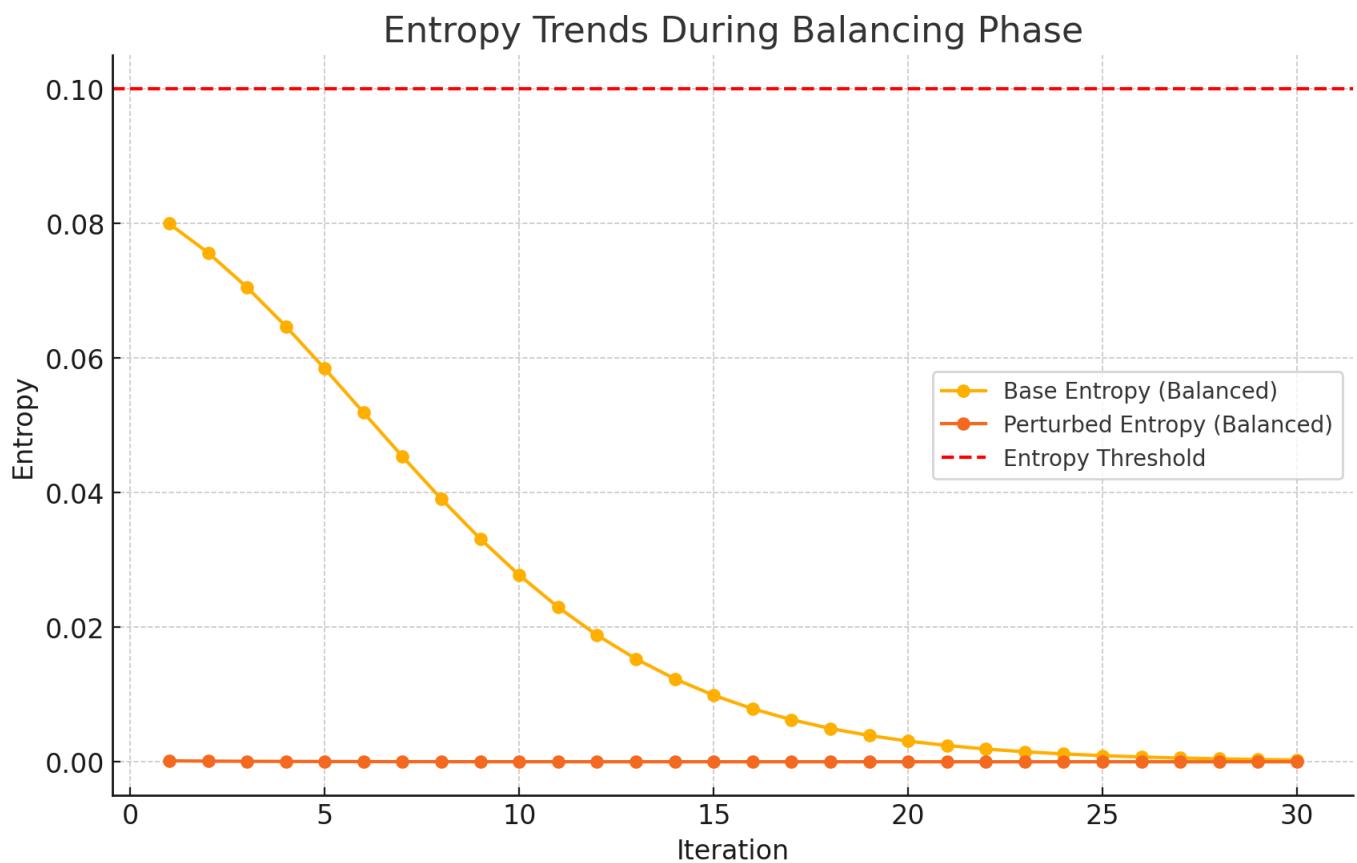
Feedback Loop Wave Morphing (Base vs Perturbed)



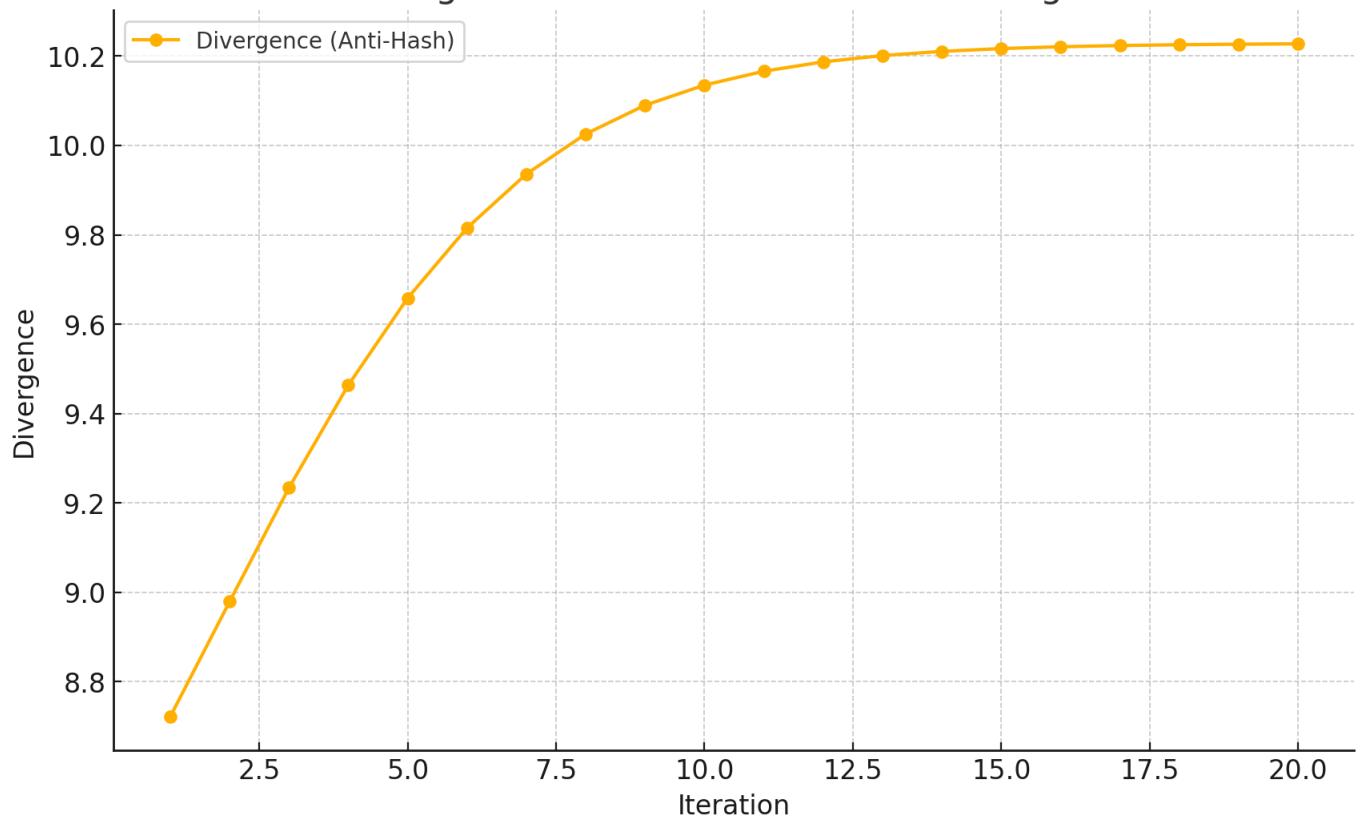
Recursive Feedback Lattices (Base vs Perturbed)

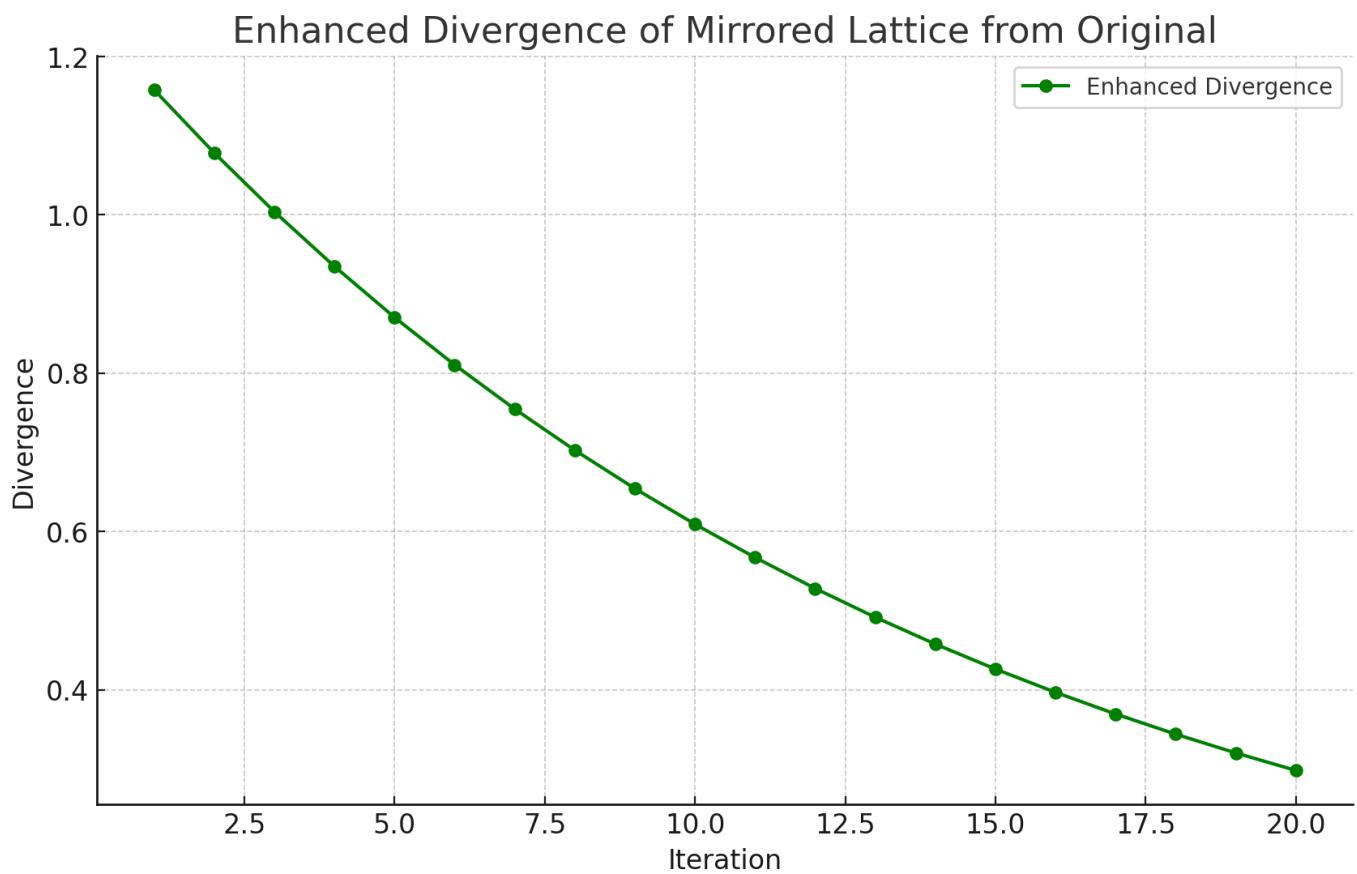




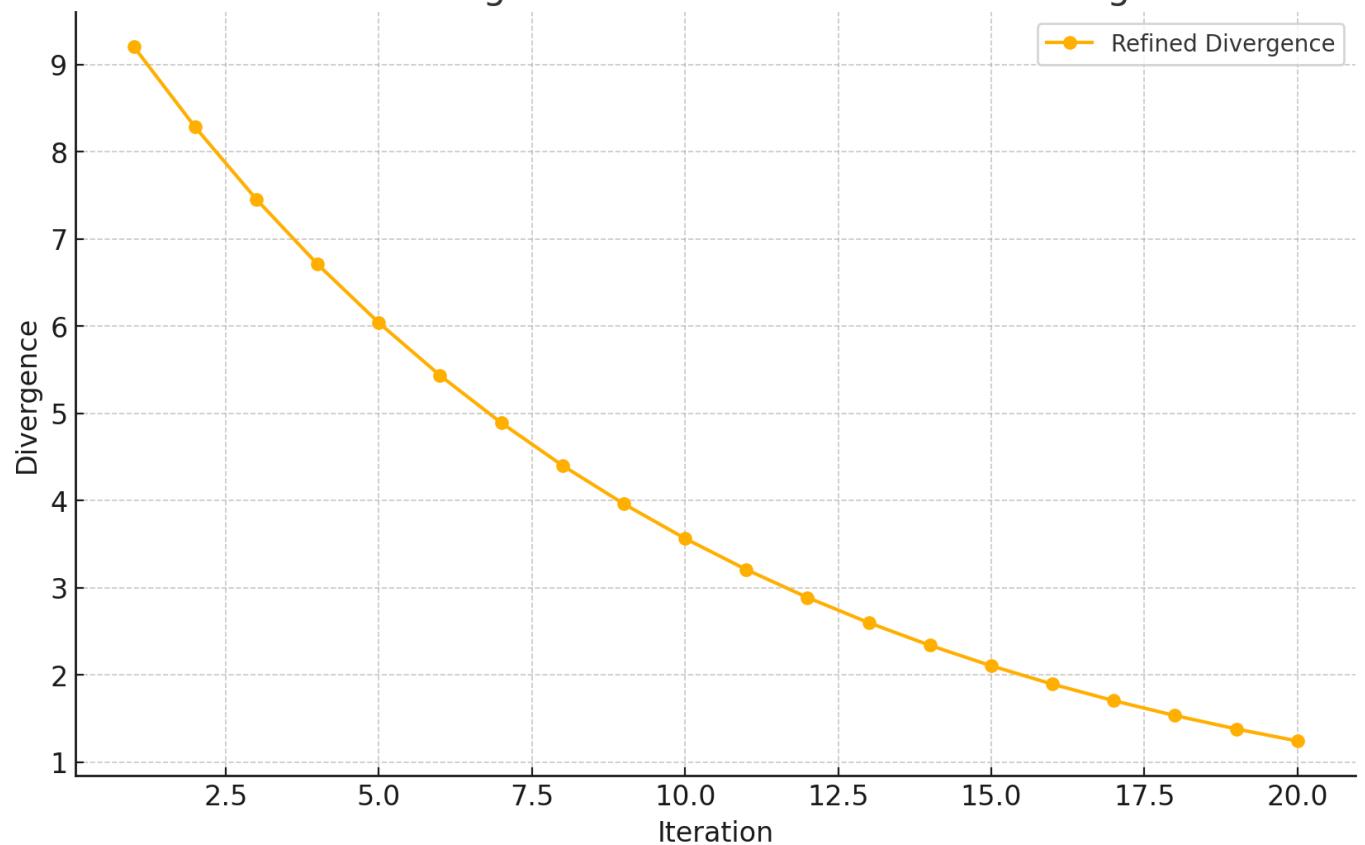


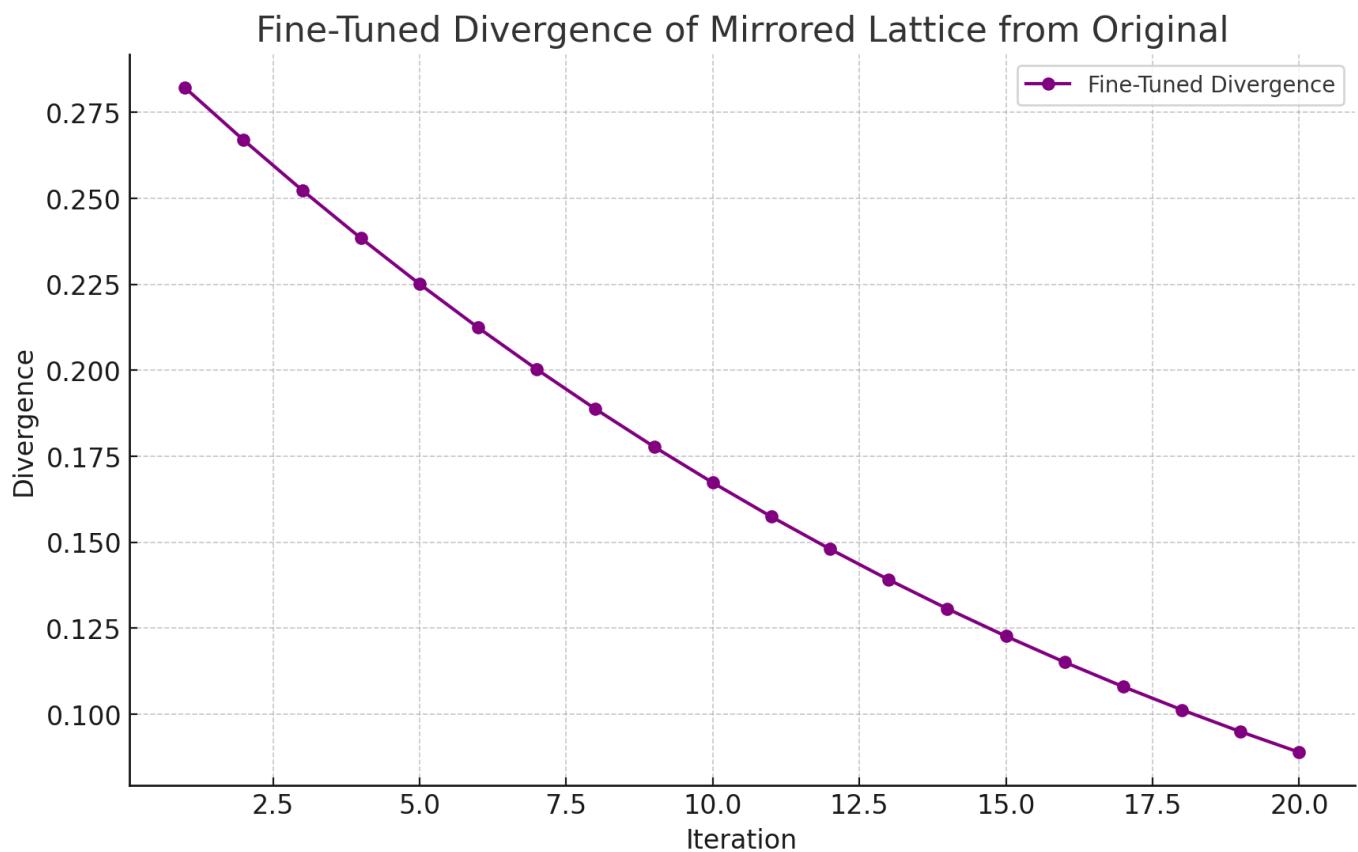
Divergence of Mirrored Lattice from Original



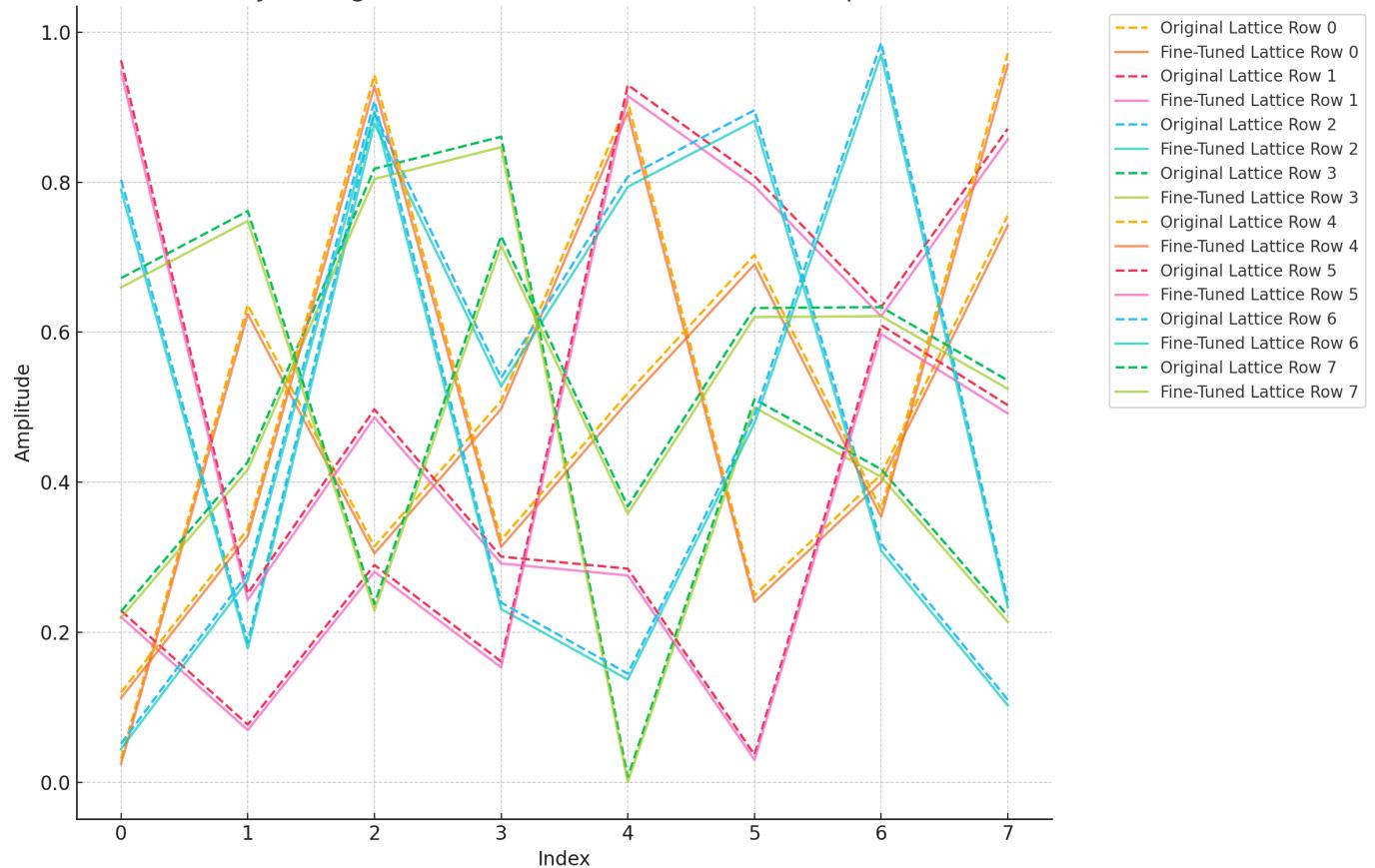


Refined Divergence of Mirrored Lattice from Original

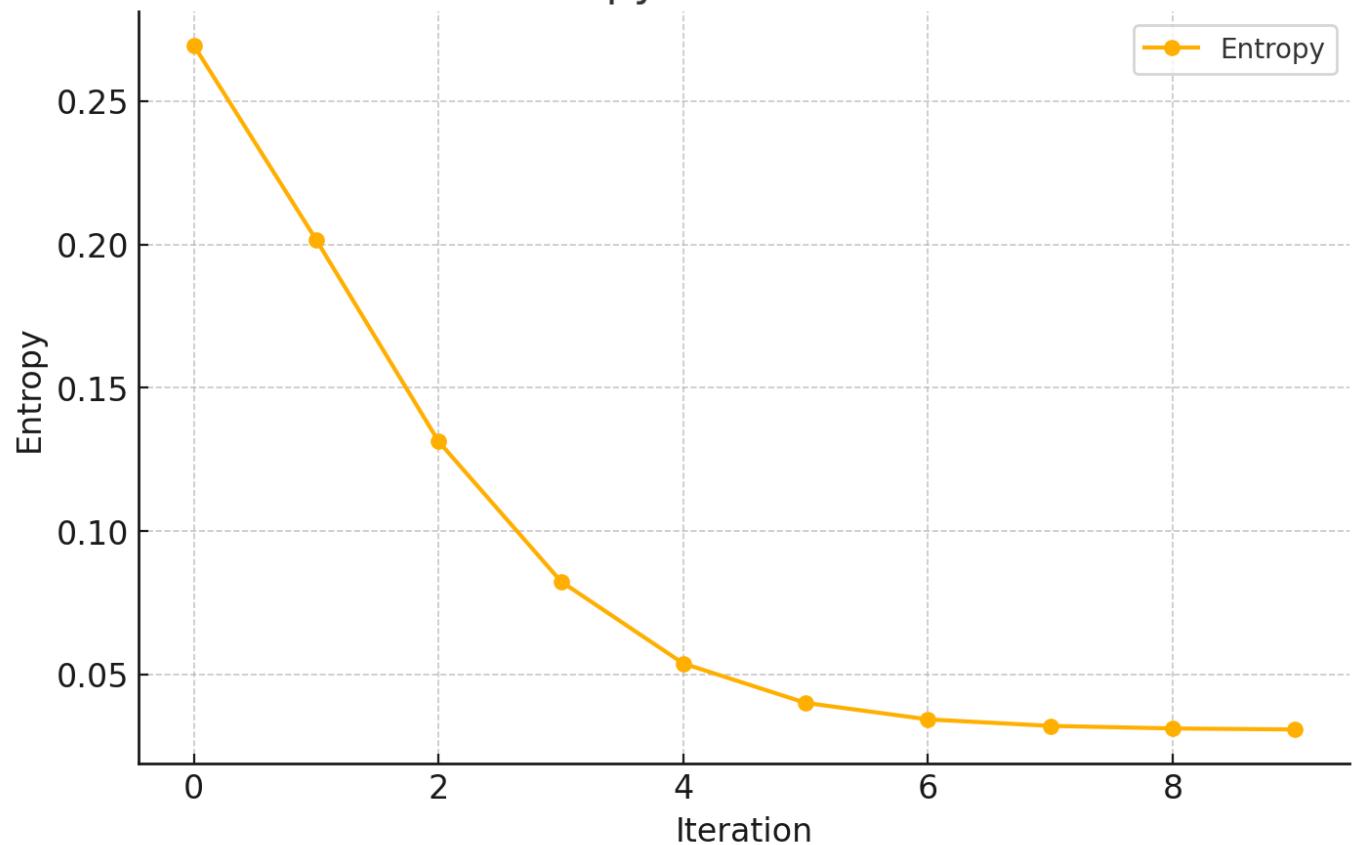




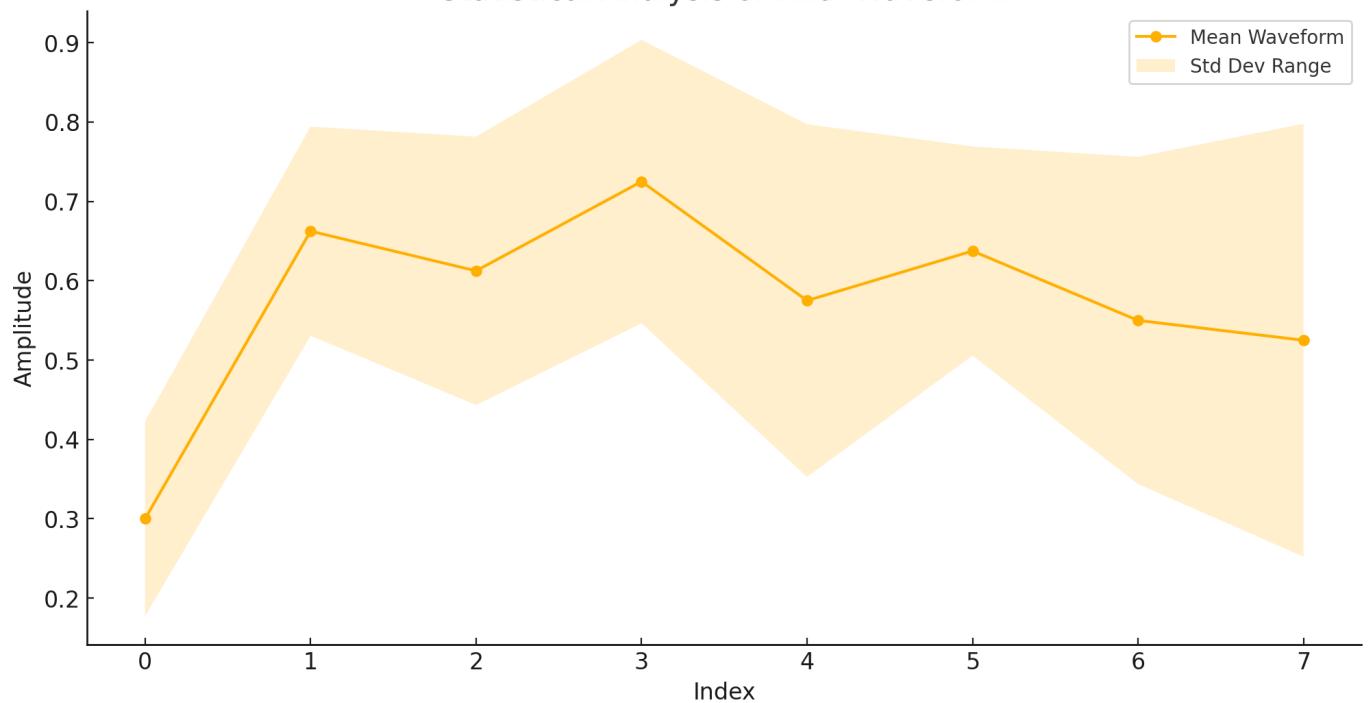
Overlay of Original and Fine-Tuned Lattices (Row Amplitudes)



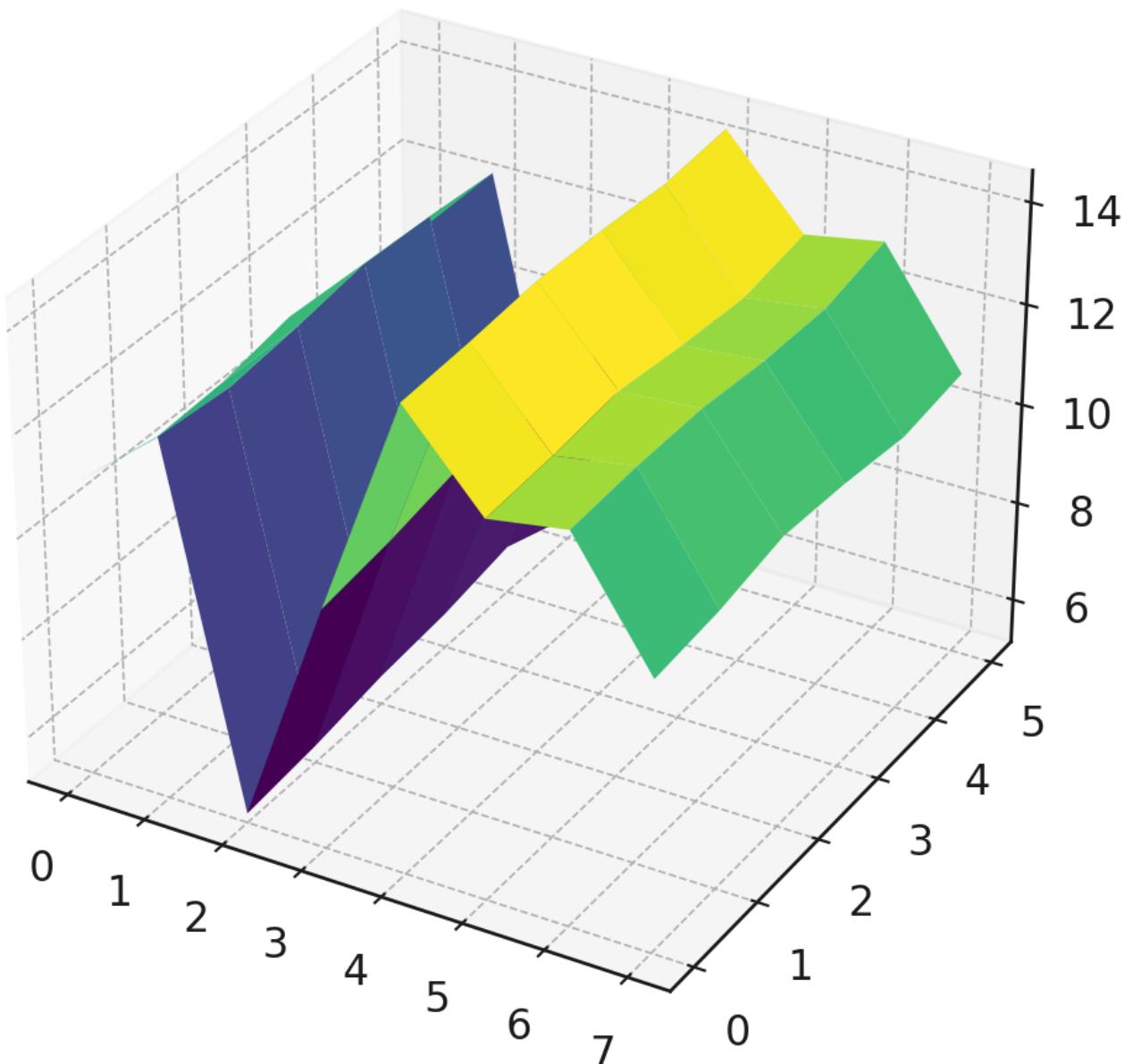
Entropy Over Iterations

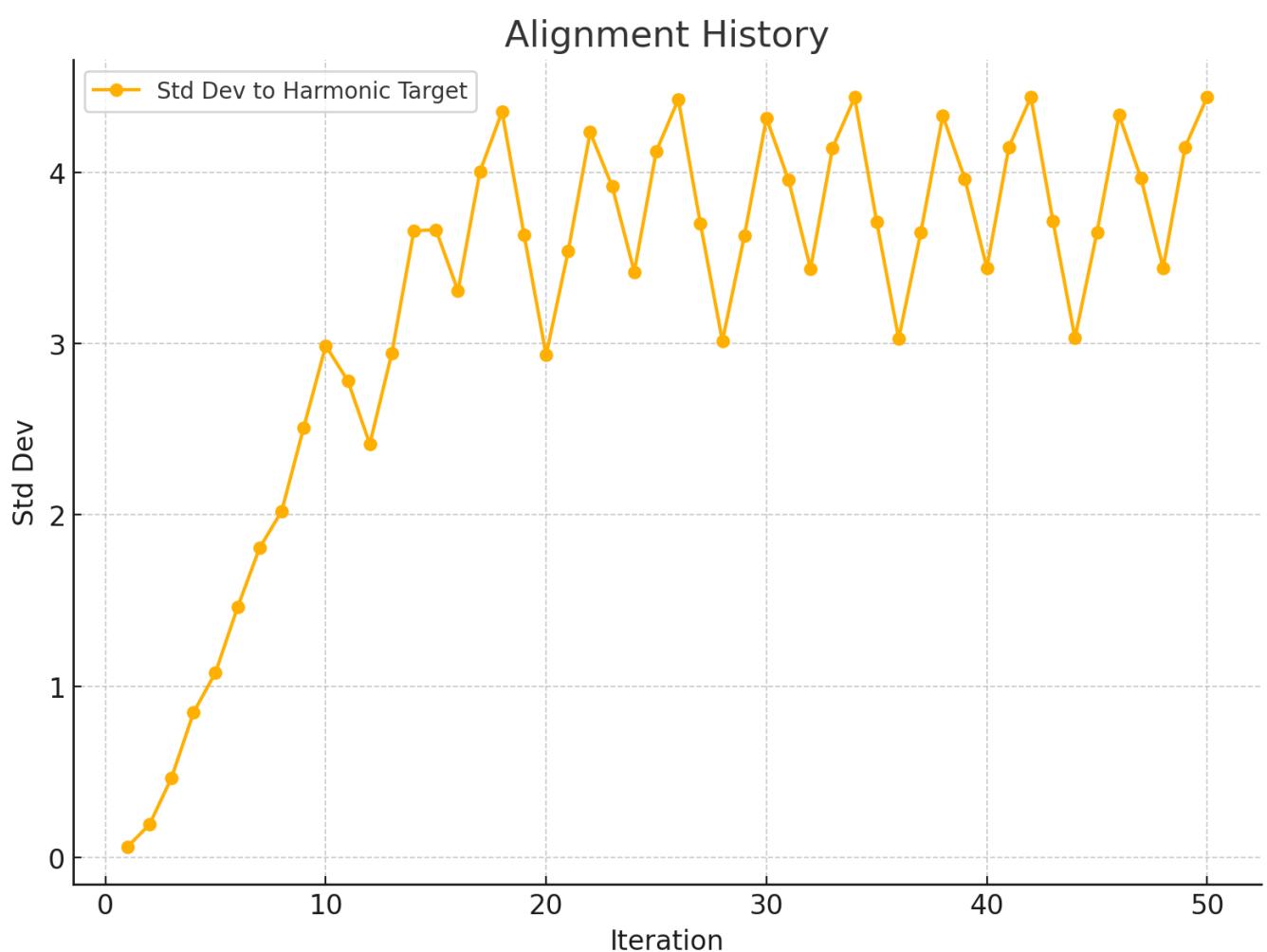
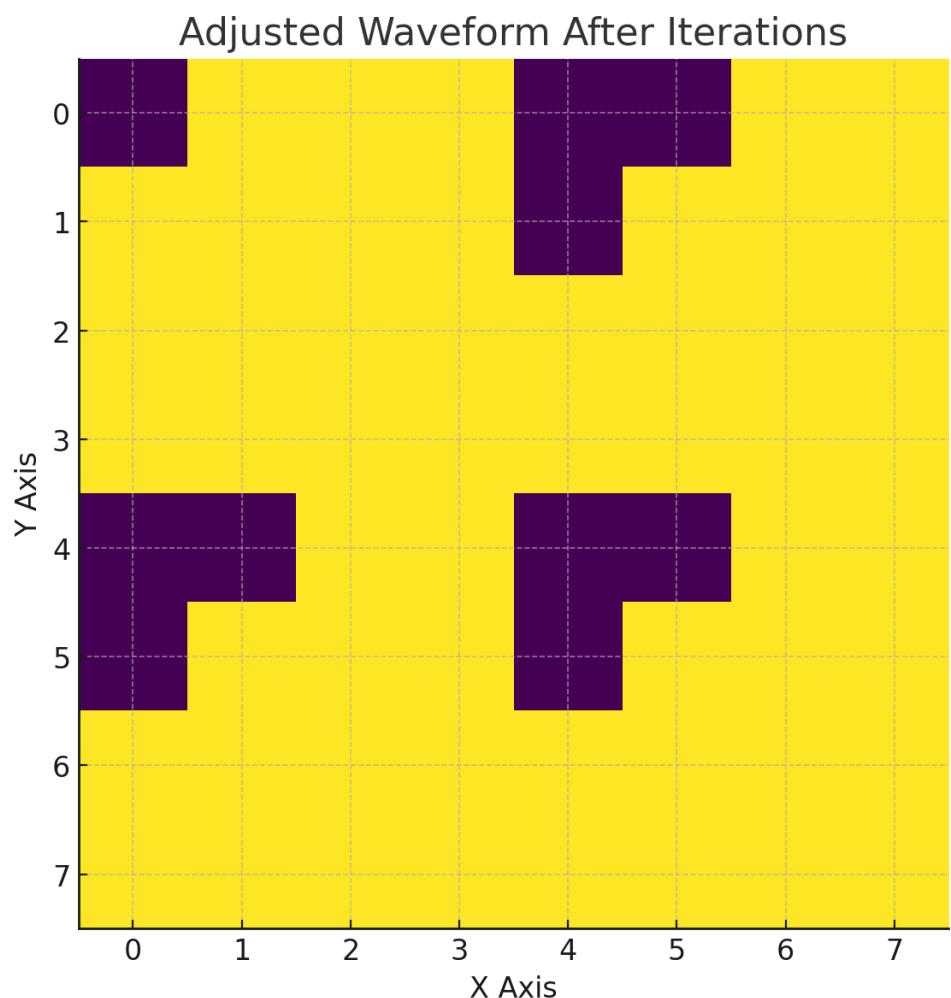


Statistical Analysis of Final Waveform

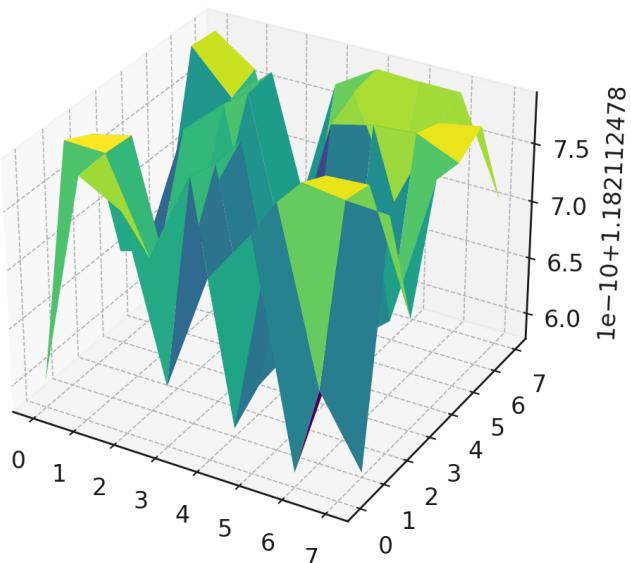


Adjusted Waveform After 50 Iterations

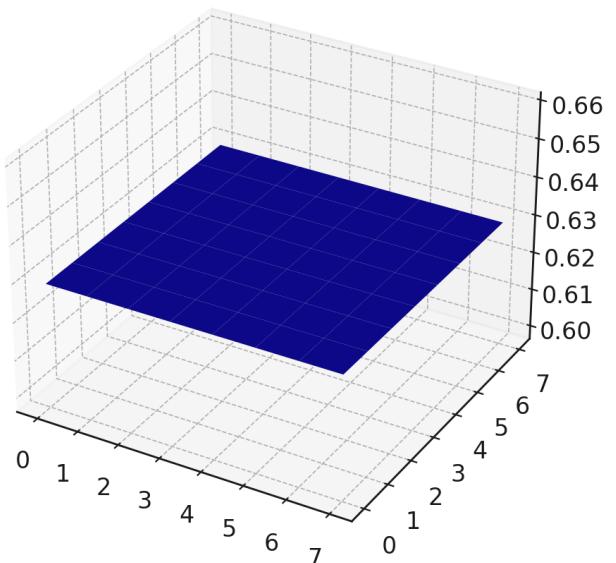




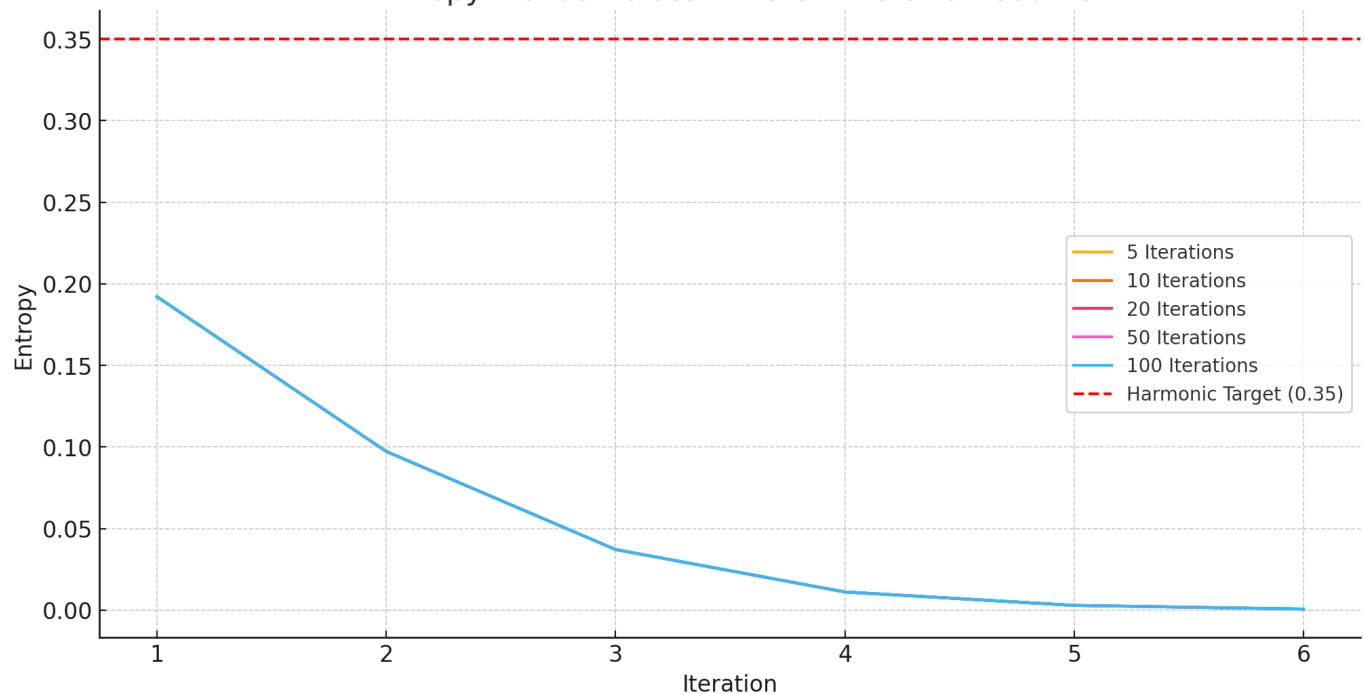
Final Waveform (Aligned with Output)

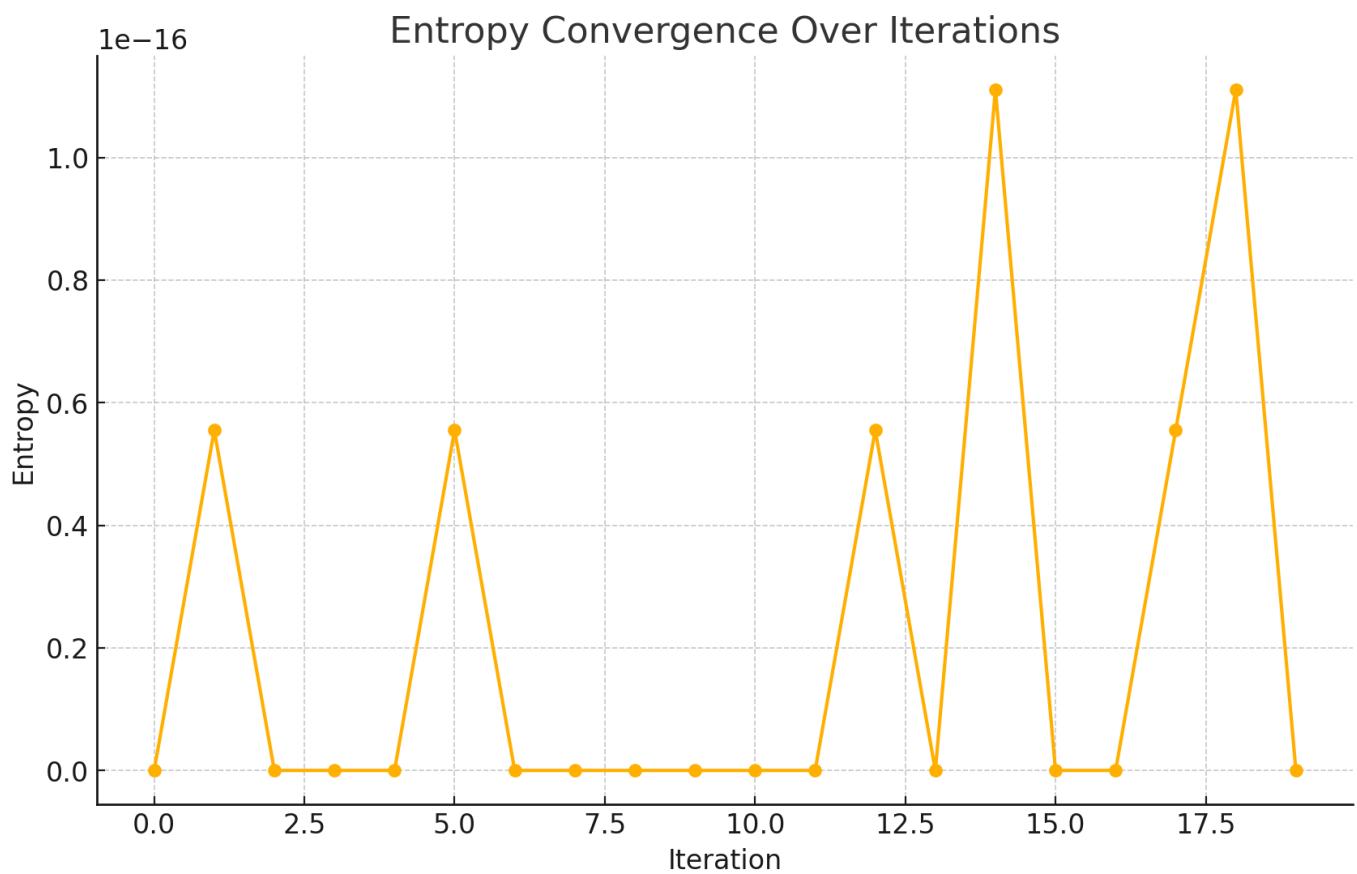


Final Padding (Waste Alignment)

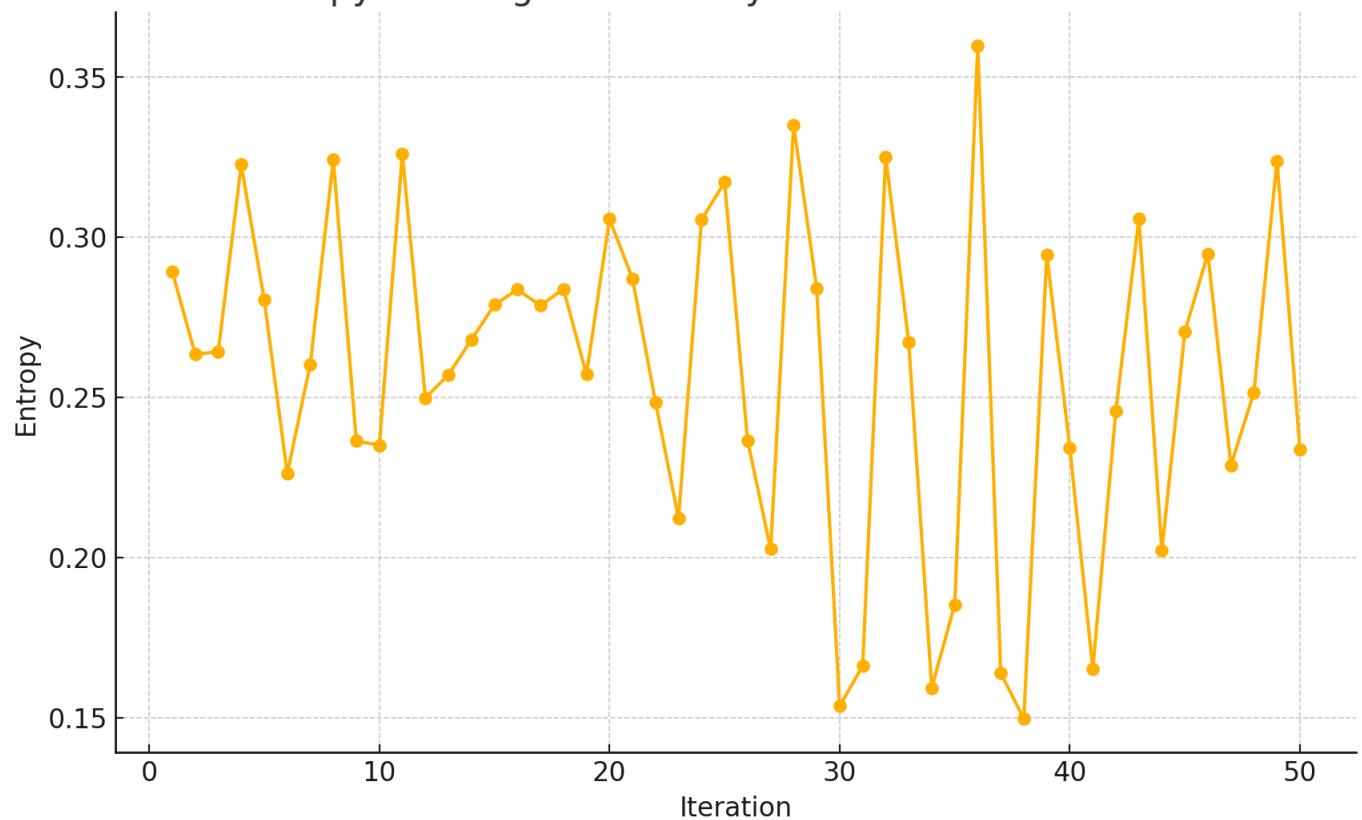


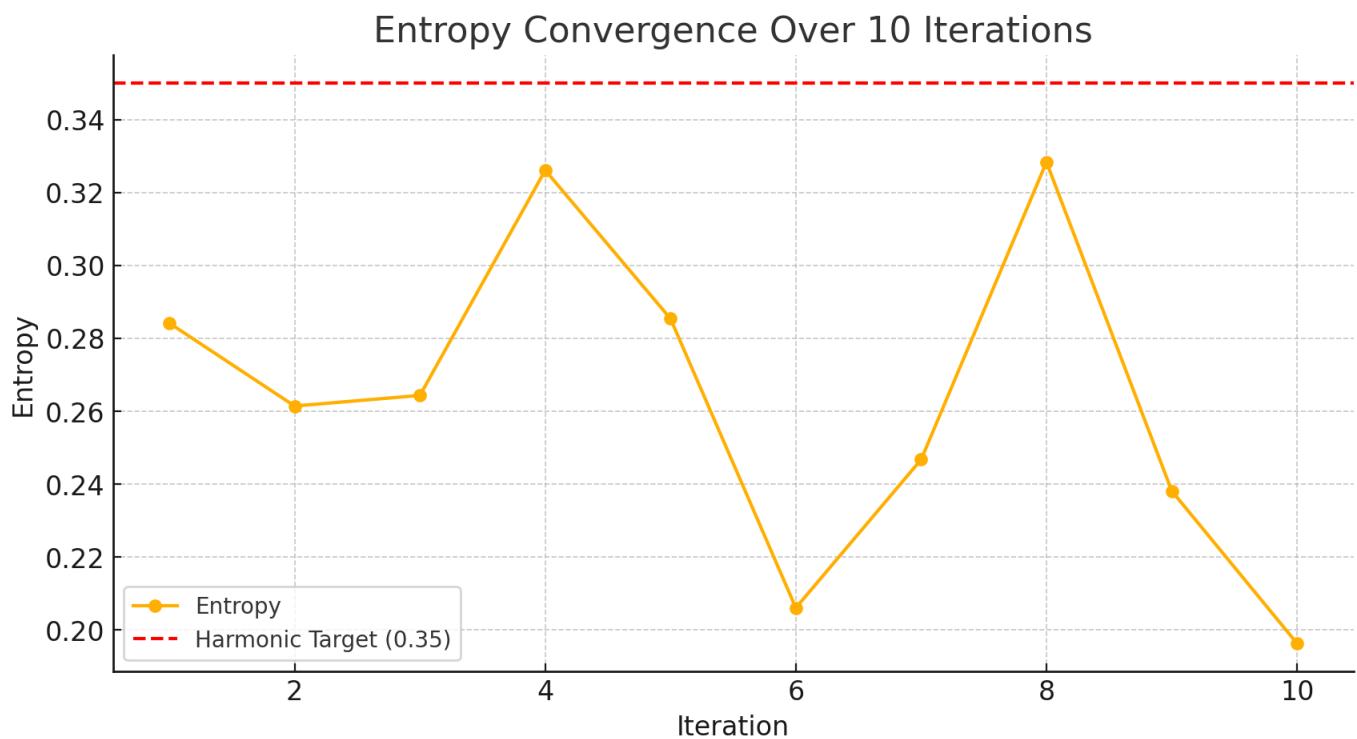
Entropy Trends Across Different Iteration Counts



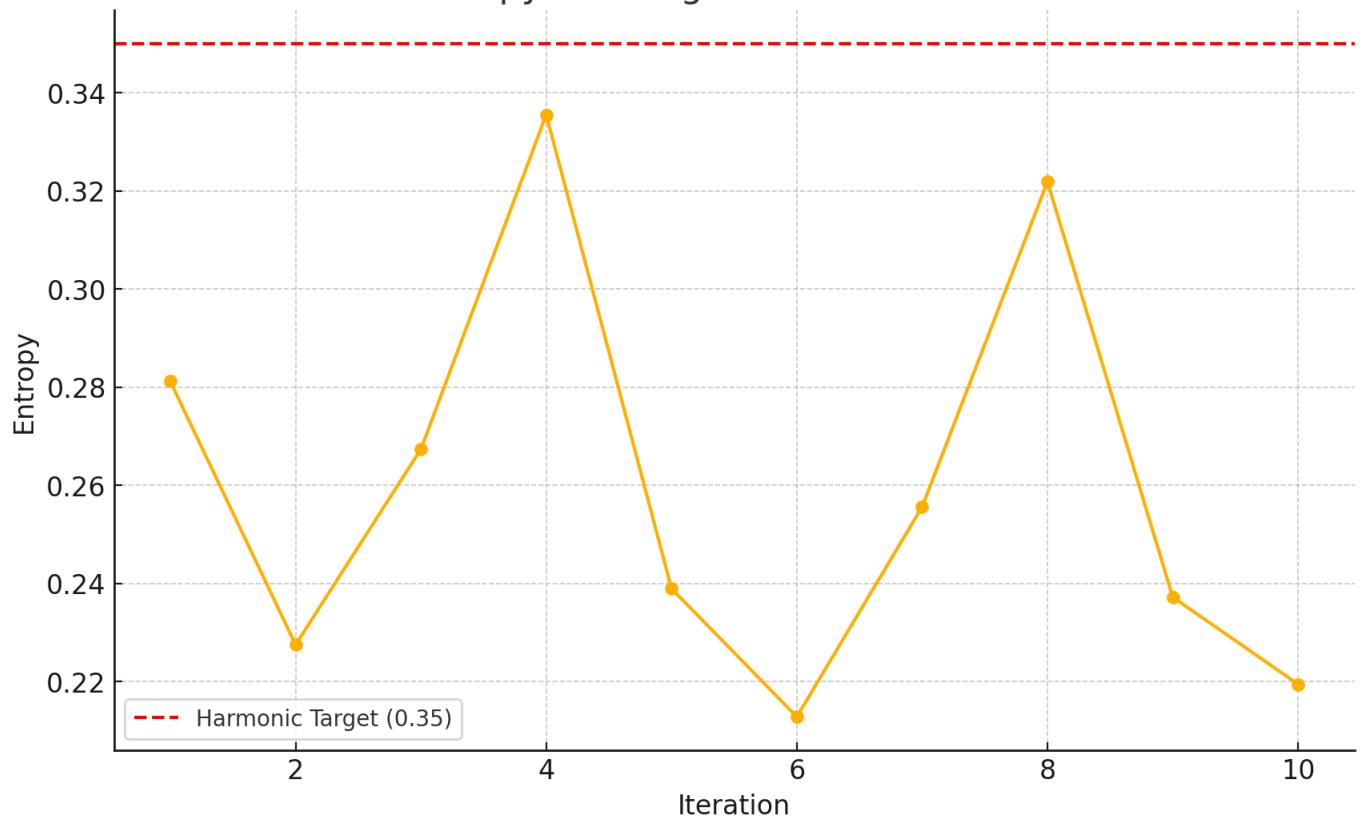


Entropy Convergence with Dynamic Harmonic Feedback

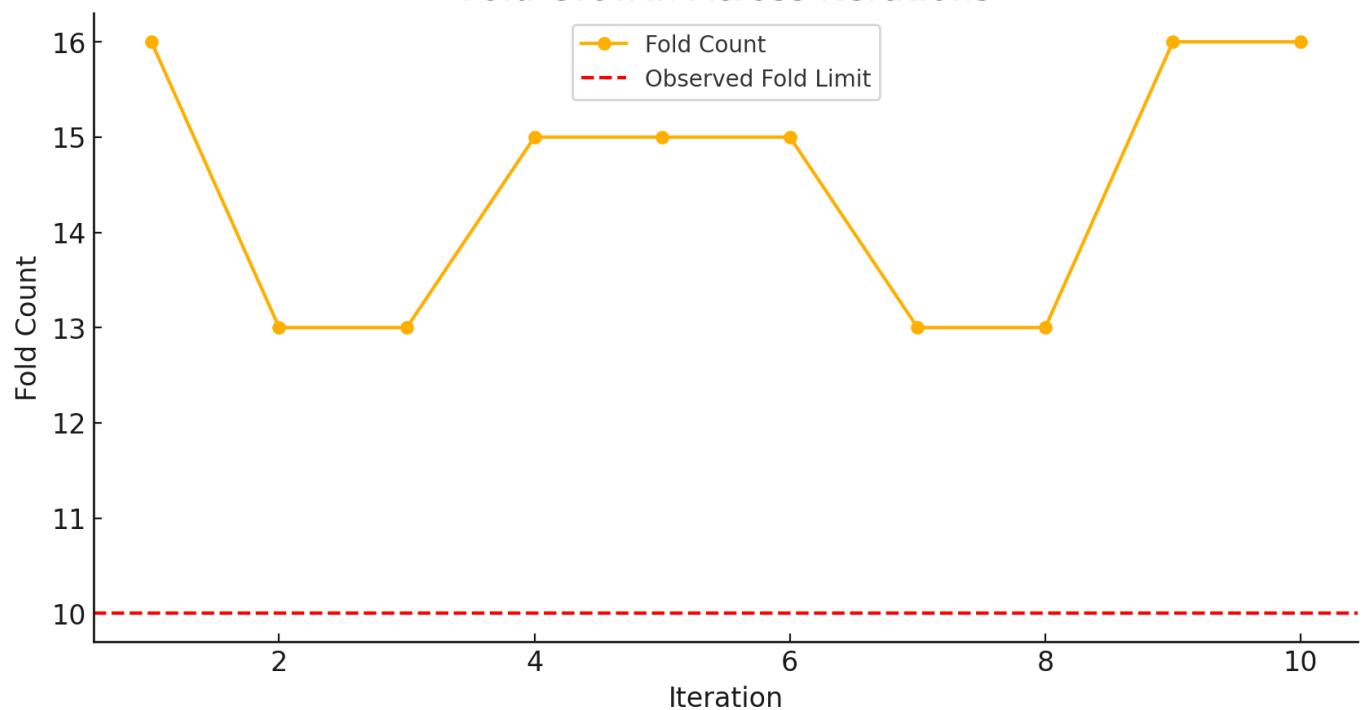




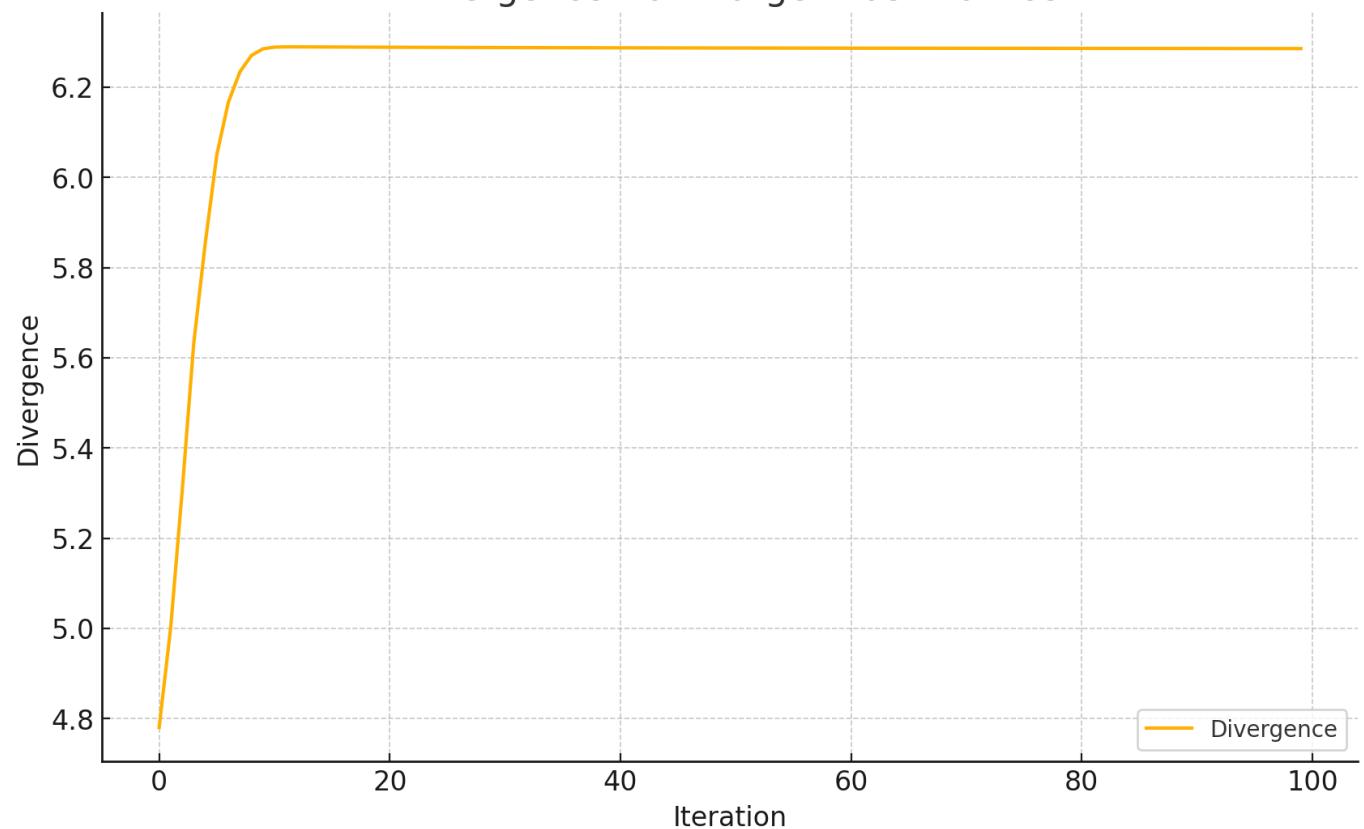
Entropy Convergence with Real Hash

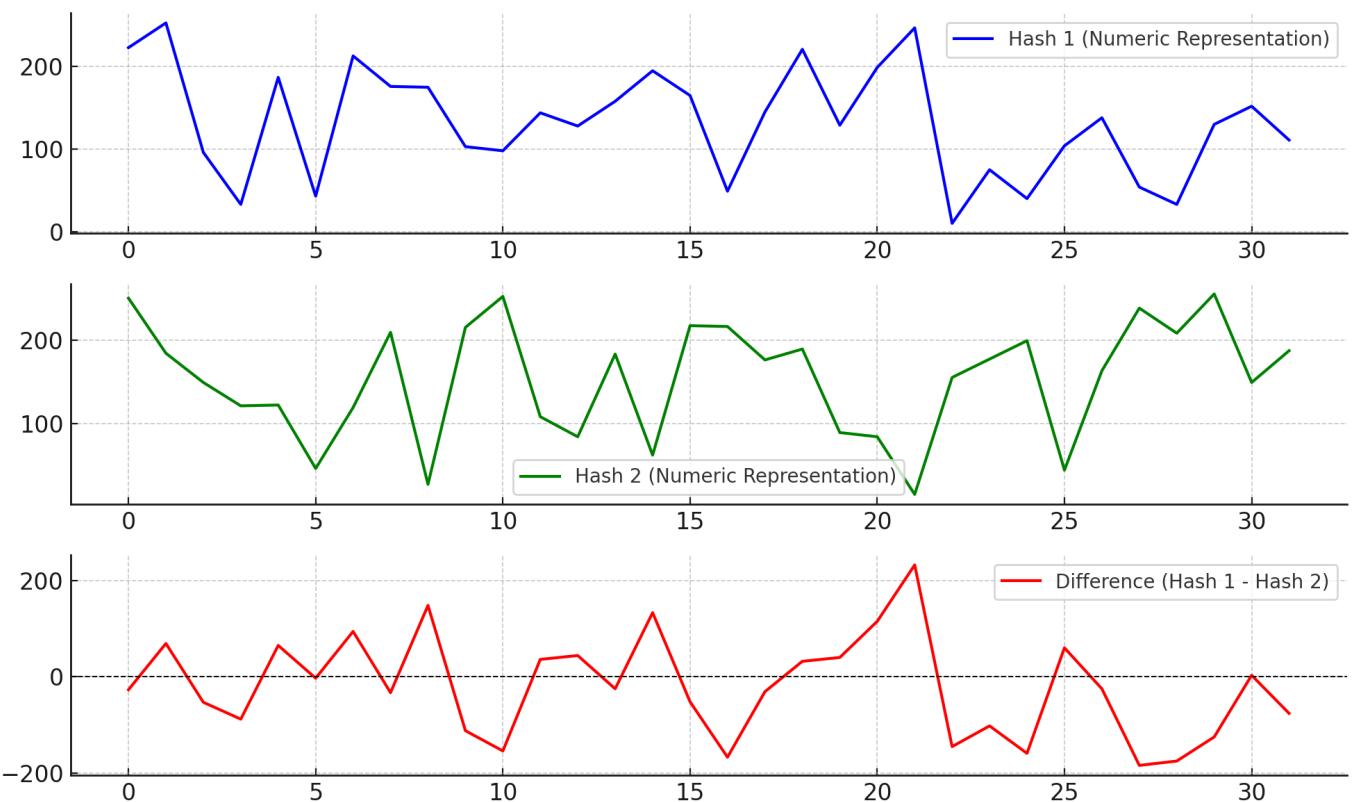


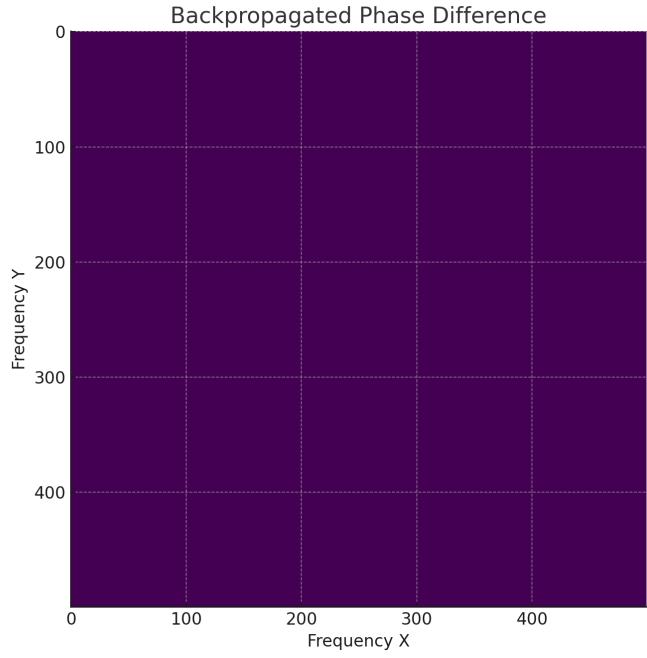
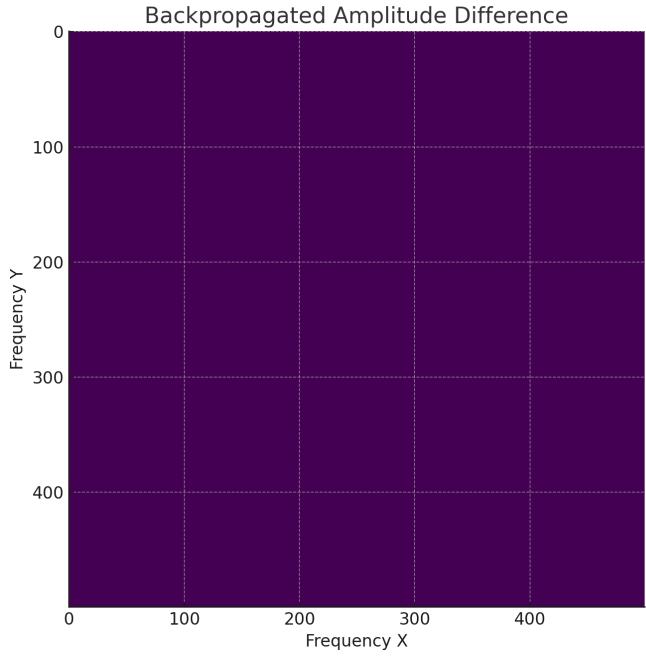
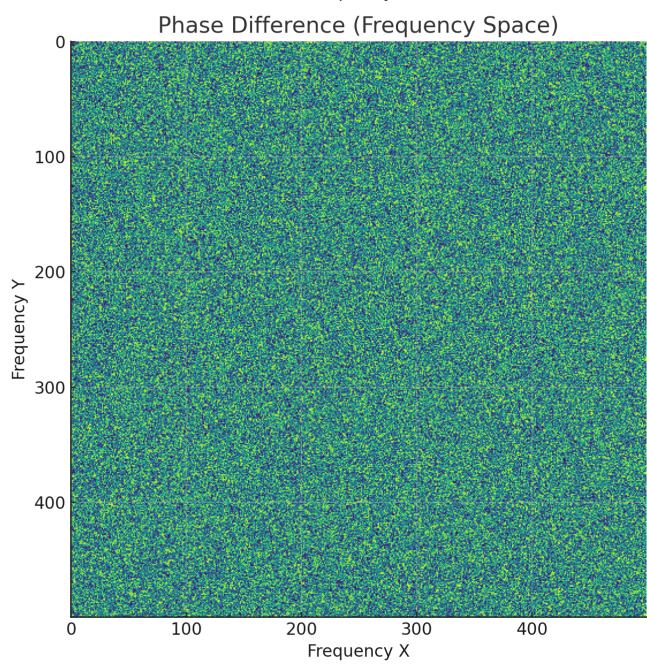
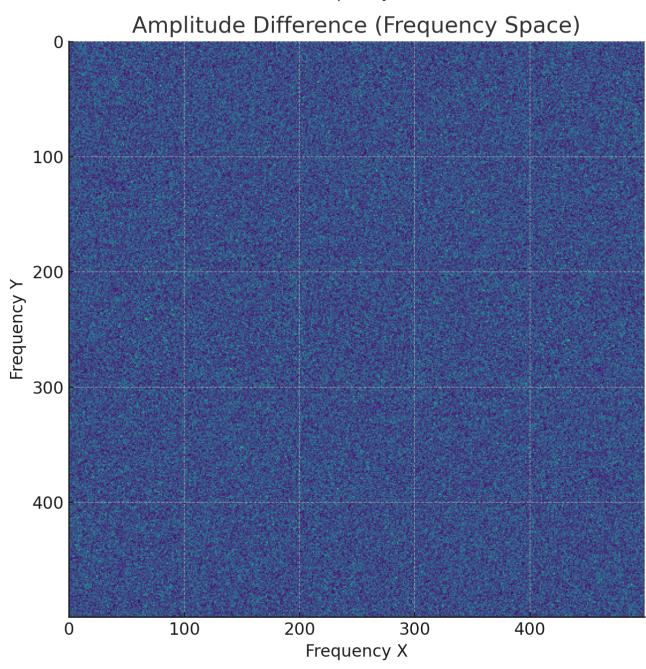
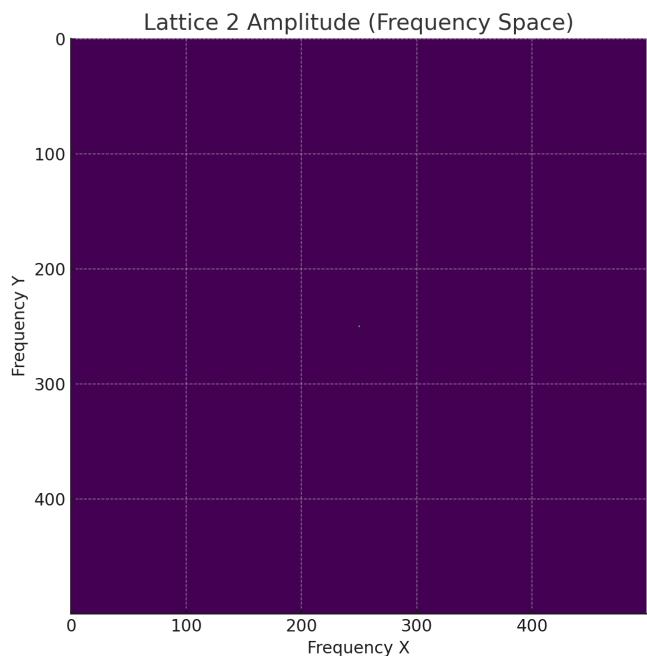
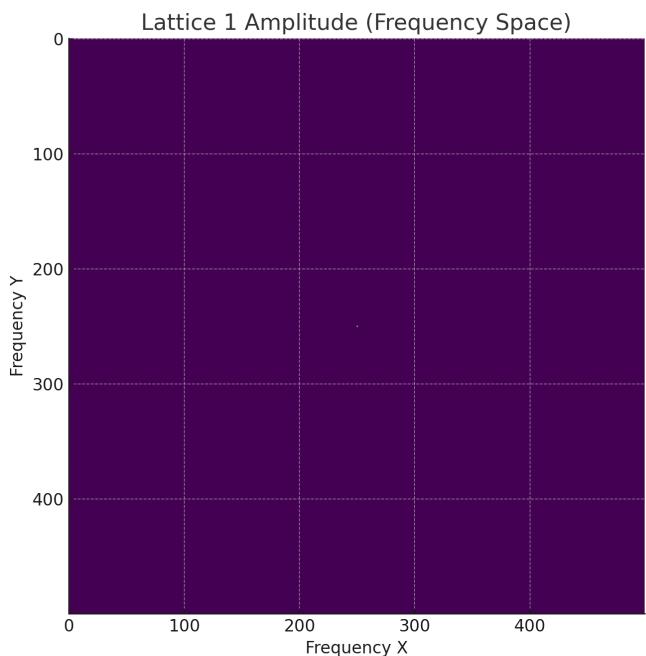
Fold Growth Across Iterations



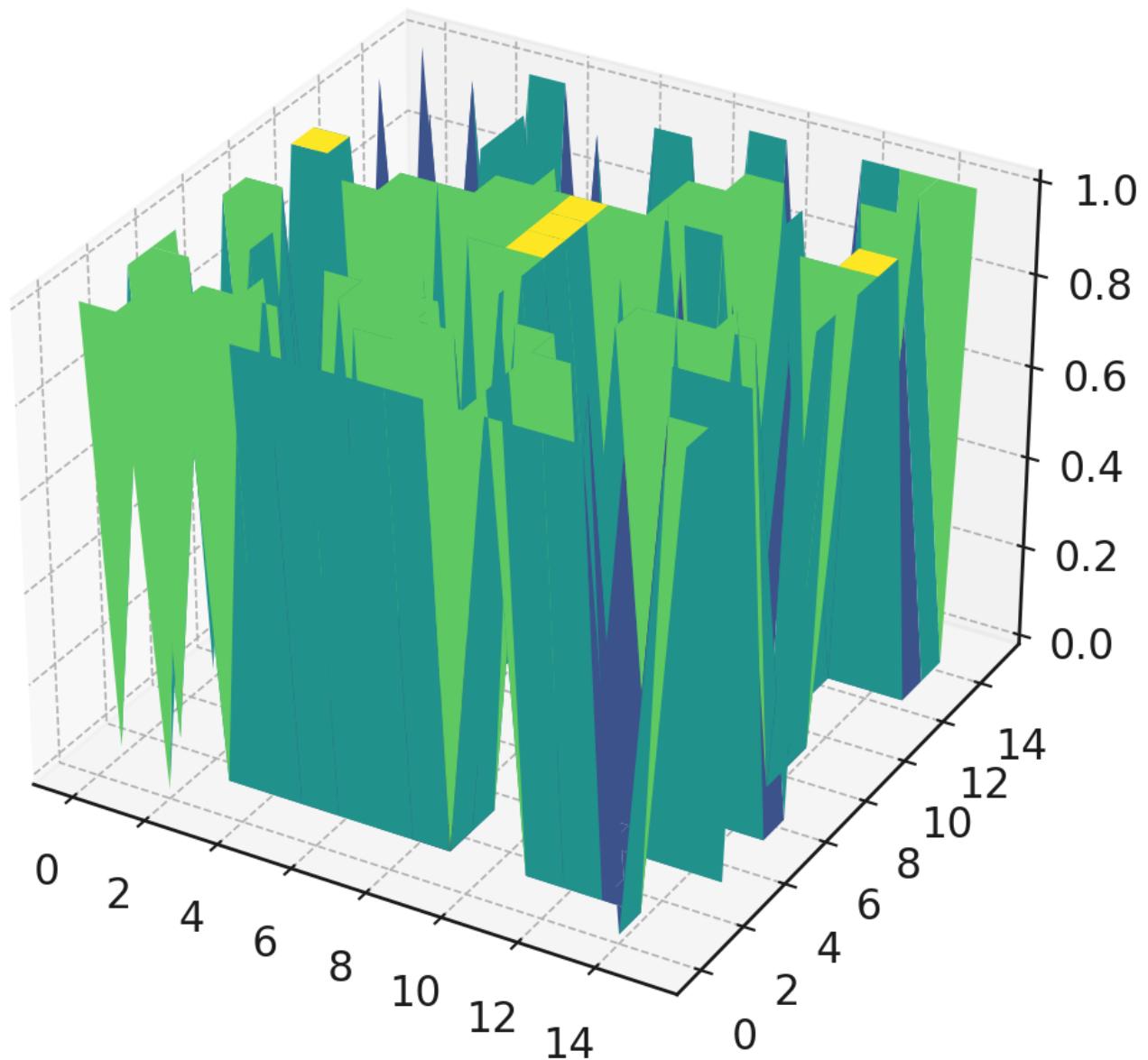
Divergence from Target Hash Lattice



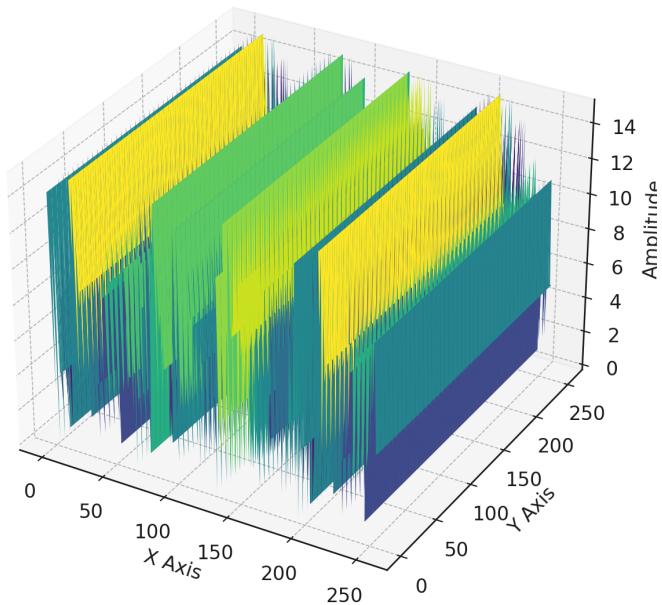




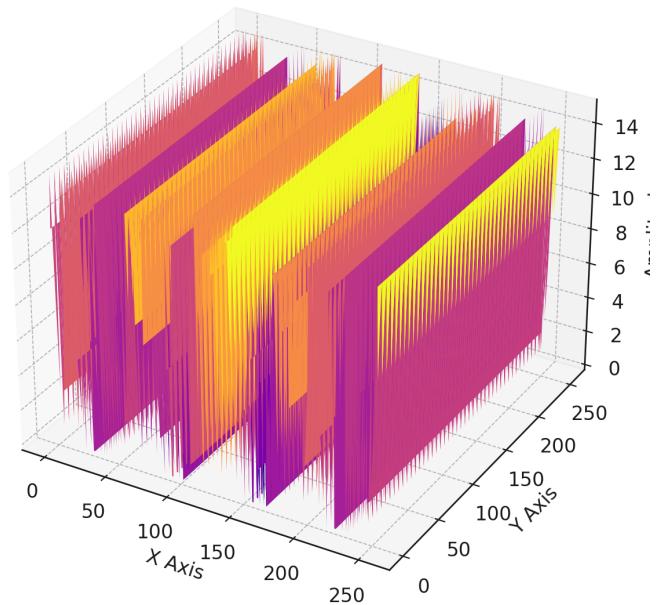
3D View of Hash 1 (Hello, World!)



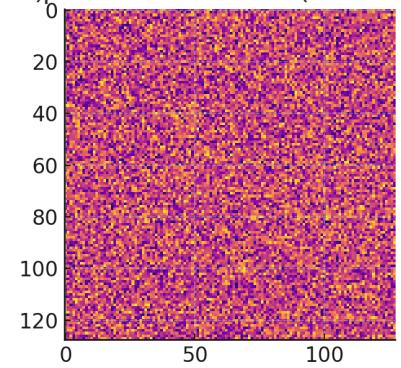
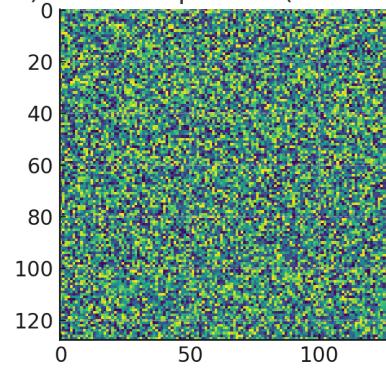
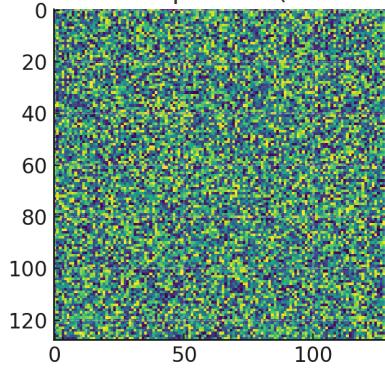
Hash 1 Lattice (256x256)



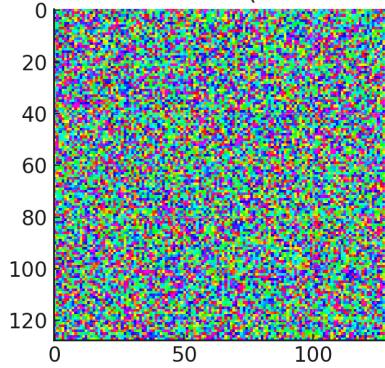
Hash 2 Lattice (256x256)



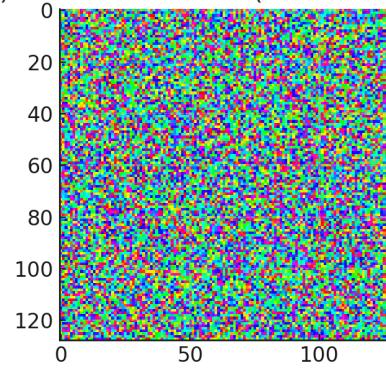
Lattice 1 Amplitude (Small Grid) Lattice 2 Amplitude (Small Grid) Amplitude Difference (Small Grid)



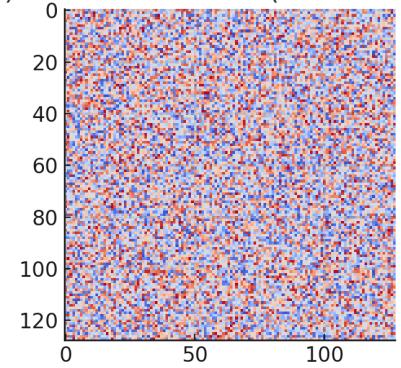
Lattice 1 Phase (Small Grid)



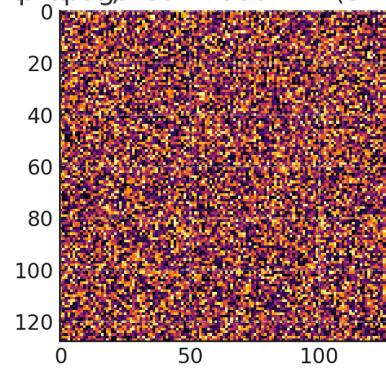
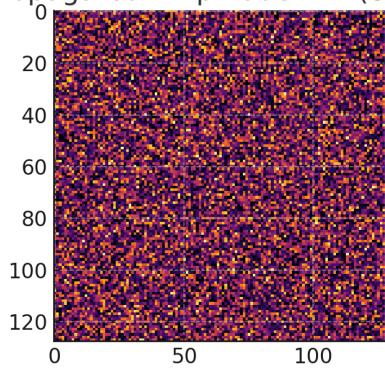
Lattice 2 Phase (Small Grid)

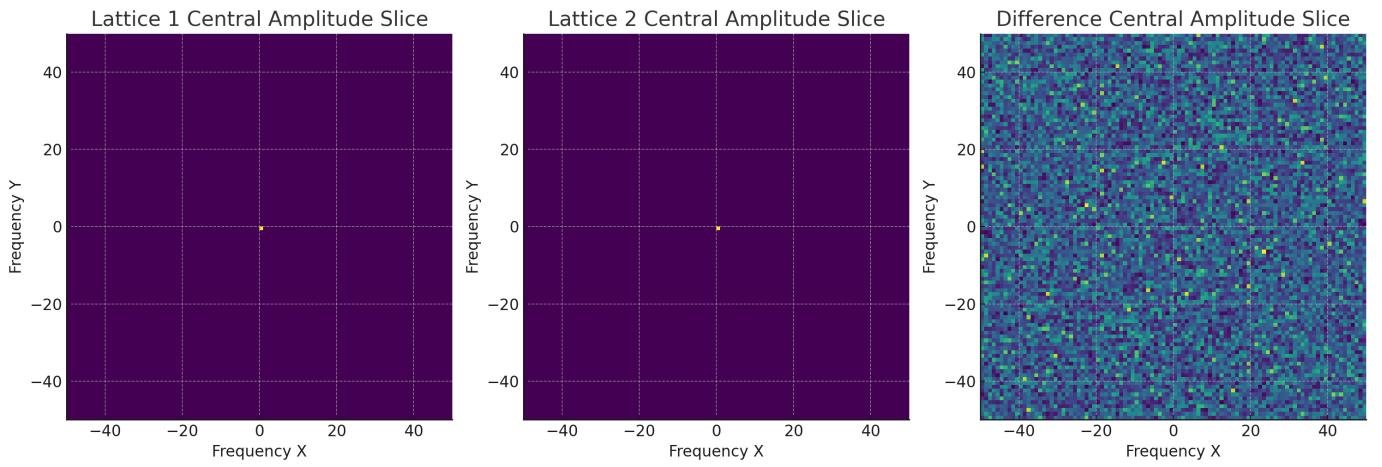


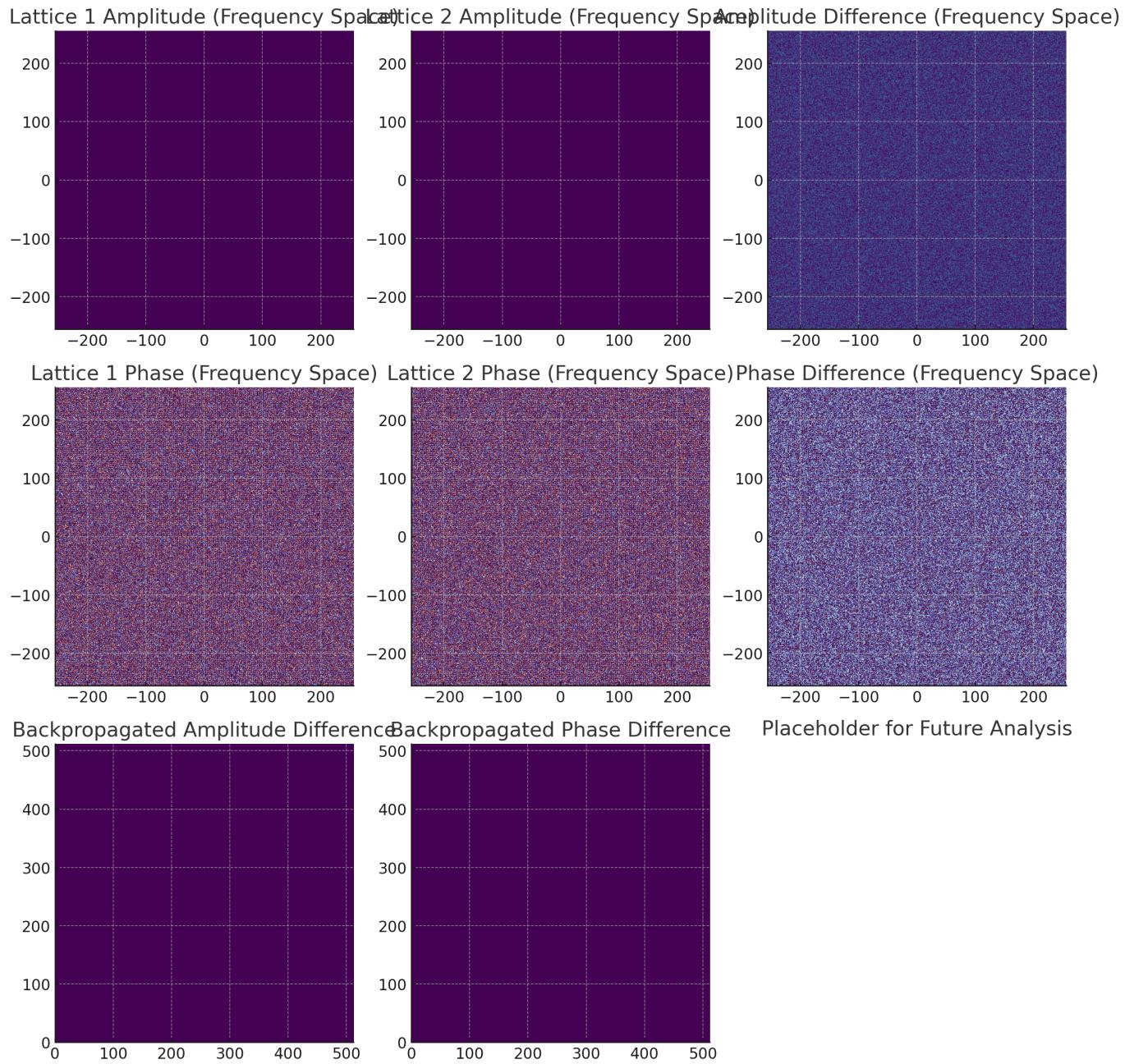
Phase Difference (Small Grid)

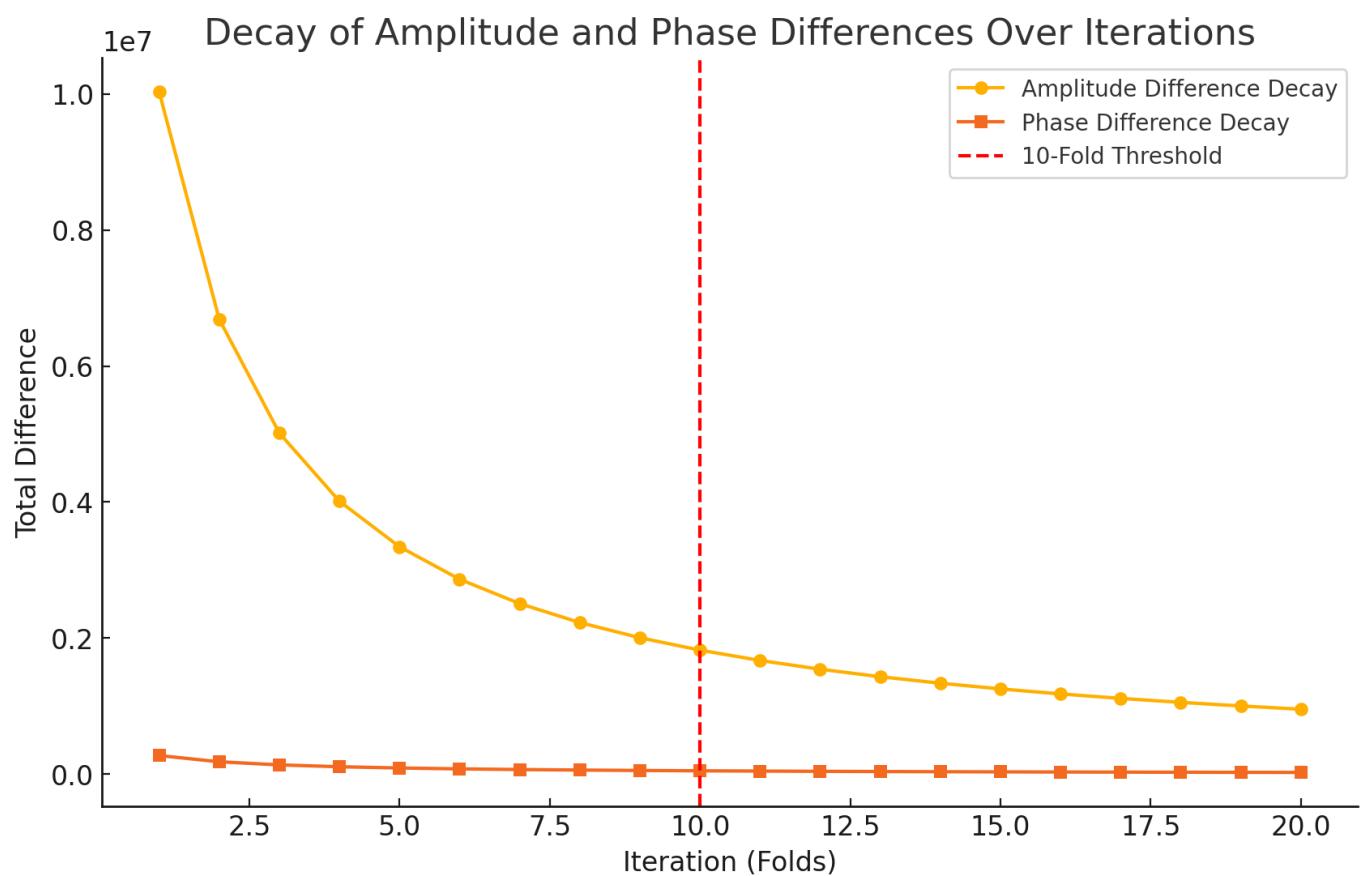


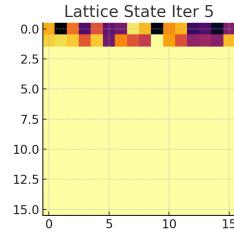
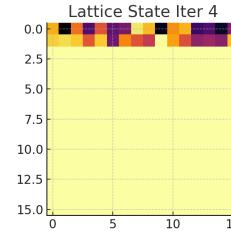
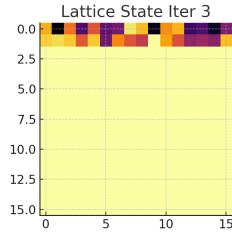
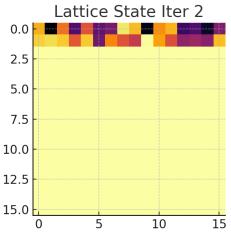
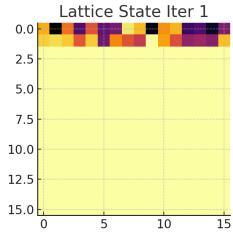
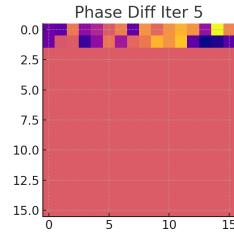
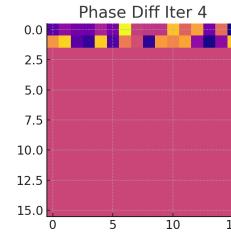
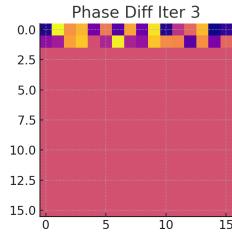
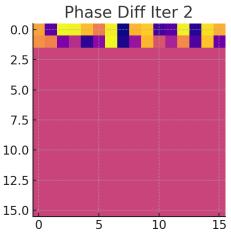
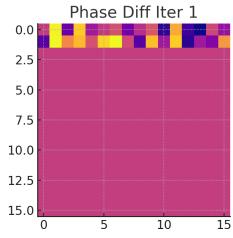
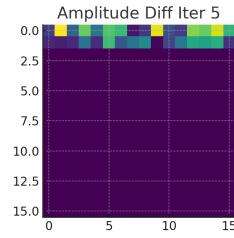
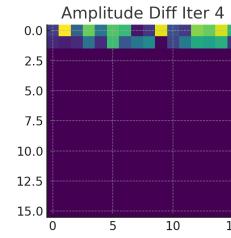
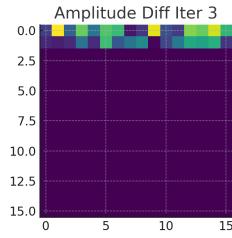
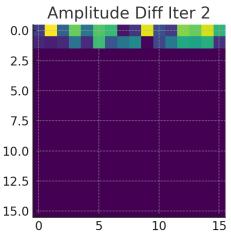
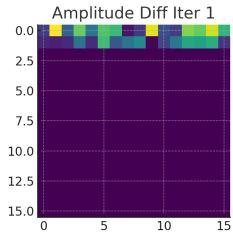
Backpropagated Amplitude Diff (Small Grid) Backpropagated Phase Diff (Small Grid)

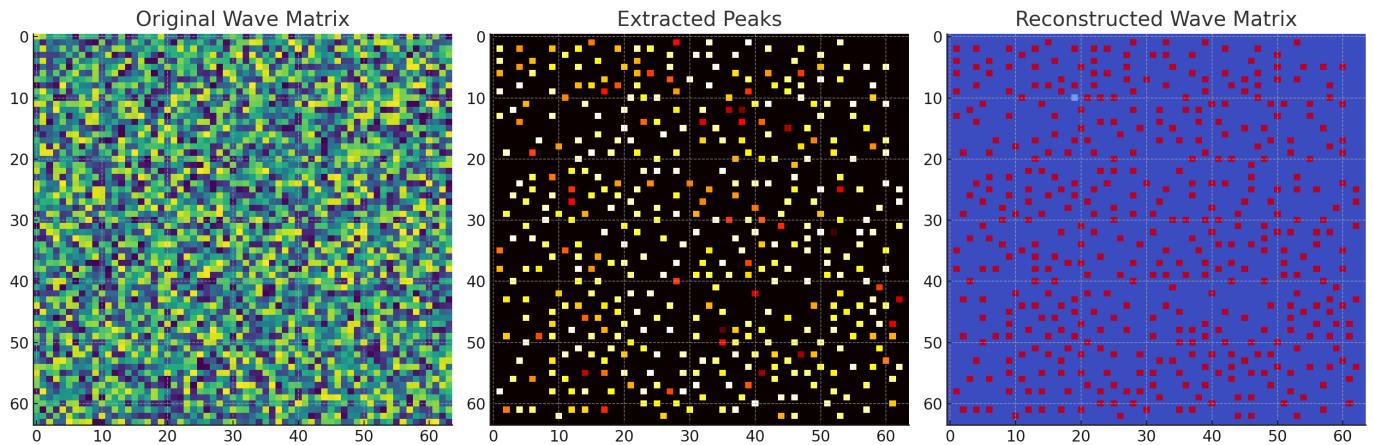












Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
import matplotlib.pyplot as plt
```

```
# Parameters for visualization
```

```
A = 2.0
```

```
B = 3.0
```

```
initial_len_c = 5.0
```

```
x = 1.0
```

```
iterations = 20
```

```
# Dynamic Len(C) function for recursive refinement
```

```
len_function = lambda current_f: max(1, current_f + 1)
```

```
# Data for visualization
```

```
values = []
```

```
current_f = 0
```

```
for i in range(iterations):
```

```
    len_c = len_function(current_f) # Calculate Len(C)
```

```
    current_f = calculate_f_cubed(A, B, len_c, x) # Update F_cubed
```

```
    values.append(current_f) # Store the value
```

```
# Plotting the results
```

```
plt.figure(figsize=(10, 6))
```

```
plt.plot(range(1, iterations + 1), values, marker='o', linestyle='-', color='b', label='F_cubed')
```

```
plt.title("Recursive Refinement of F_cubed Over Iterations", fontsize=14)
```

```
plt.xlabel("Iteration", fontsize=12)
```

```
plt.ylabel("F_cubed Value", fontsize=12)
```

```
plt.grid(True, linestyle='--', alpha=0.6)
```

```
plt.legend(fontsize=12)
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

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

# Define the function to calculate F_cubed with twist and spin contributions
def calculate_f_cubed_with_axes(A, B, len_c, x, axis_weights):
    # Quadratic component for twist and spin
    twist = axis_weights[0] * (A**2) # Twist driven by axis 1
    spin = axis_weights[1] * (B**2) # Spin driven by axis 2

    # Exponential torque component
    torque = axis_weights[2] * (1 + math.exp(-10 * (A * x - 0.35)))

    # Combine all components
    return (twist + spin) * len_c * torque

# Recursive refinement function to calculate F_cubed with dynamic contributions
def recursive_f_cubed_with_axes(A, B, len_function, x, iterations):
    current_f = 0 # Initialize F_cubed value
    axis_weights = [1, 1, 1] # Initial weights for axes
    values = [] # Store overall F_cubed values
    axis_contributions = [] # Store individual axis contributions

    for i in range(iterations):
        len_c = len_function(current_f) # Calculate Len(C)
        axis_weights = [np.sin(i / 10), np.cos(i / 10), np.sin(i / 20)] # Dynamic twist, spin, torque weights
        current_f = calculate_f_cubed_with_axes(A, B, len_c, x, axis_weights) # Update F_cubed

        values.append(current_f) # Store overall value
        axis_contributions.append(axis_weights) # Store current axis weights
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
import matplotlib.pyplot as plt
```

```
# Extract distances and relations
```

```
distances = [r['Distance'] for r in results if r['Distance'] is not None]
```

```
relations = [r['Relation'] for r in results if r['Relation'] is not None]
```

```
# Plot distances and relations
```

```
plt.figure(figsize=(12, 6))
```

```
# Distances
```

```
plt.subplot(2, 1, 1)
```

```
plt.plot(range(1, len(distances) + 1), distances, label='Distances', color='blue')
```

```
plt.title('Distances Between Constants')
```

```
plt.xlabel('Index')
```

```
plt.ylabel('Distance')
```

```
plt.grid(True)
```

```
plt.legend()
```

```
# Relations
```

```
plt.subplot(2, 1, 2)
```

```
plt.plot(range(1, len(relations) + 1), relations, label='Relations', color='green')
```

```
plt.title('Relations Between Constants')
```

```
plt.xlabel('Index')
```

```
plt.ylabel('Relation')
```

```
plt.grid(True)
```

```
plt.legend()
```

```
plt.tight_layout()
```

```
plt.show()
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
import matplotlib.pyplot as plt
```

```
# Constants defined by the SHA-256 algorithm
```

```
K = [
```

```
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
    0xe49b69c1, 0xefbe4786, 0xfc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
    0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
    0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90beffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
]
```

```
# Calculate differences and ratios between constants
```

```
differences = [K[i] - K[i - 1] for i in range(1, len(K))]
```

```
ratios = [K[i] / K[i - 1] for i in range(1, len(K))]
```

```
# Plot the constants, differences, and ratios
```

```
plt.figure(figsize=(15, 10))
```

```
# Plot constants
```

```
plt.subplot(3, 1, 1)
```

```
plt.plot(range(len(K)), K, label="Constants (K)", color="blue", marker="o")
```

```
plt.title("SHA-256 Constants (K)")
```

```
plt.xlabel("Index")
```

```
plt.ylabel("Value")
```

```
plt.grid(True)
```

```
plt.legend()
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
import numpy as np
```

```
# Define a mapping function to approximate ASM instruction alignment
def map_constants_to_asm(K):
    asm_operations = []
    for i in range(len(K)):
        if i > 0:
            delta = K[i] - K[i - 1]
            if delta > 0:
                asm_operations.append(f"add {delta:#x}") # Represent delta as a positive addition
            elif delta < 0:
                asm_operations.append(f"sub {abs(delta):#x}") # Represent delta as a subtraction
            else:
                asm_operations.append("nop") # No change
        else:
            asm_operations.append(f"mov eax, {K[i]:#x}") # Initialize with the first constant
    return asm_operations

# Map constants to ASM-like instructions
asm_instructions = map_constants_to_asm(K)

# Generate a waveform representation from the constants
waveform_x = np.linspace(0, len(K) - 1, len(K))
waveform_y = np.array(K) / max(K) # Normalize for visualization

# Plot the waveform
plt.figure(figsize=(12, 6))
plt.plot(waveform_x, waveform_y, label="SHA-256 Waveform", color="blue")
plt.title("Waveform Derived from SHA-256 Constants")
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
from scipy.fft import fft
```

```
# 1. Analyze the harmonic properties of the waveform (Fourier Analysis)
```

```
# Compute the Fourier Transform of the waveform
```

```
frequencies = np.fft.fftfreq(len(waveform_y), d=1) # Frequency components
```

```
fft_values = fft(waveform_y) # Fourier transform
```

```
# Plot the magnitude spectrum
```

```
plt.figure(figsize=(12, 6))
```

```
plt.plot(frequencies[:len(frequencies)//2], np.abs(fft_values)[:len(fft_values)//2], label="Magnitude Spectrum", color="purple")
```

```
plt.title("Fourier Analysis of SHA-256 Waveform")
```

```
plt.xlabel("Frequency")
```

```
plt.ylabel("Magnitude")
```

```
plt.grid(True)
```

```
plt.legend()
```

```
plt.show()
```

```
# 2. Simulate kinetic behavior using ASM instructions
```

```
# For simplicity, simulate a value progressing through the operations
```

```
def simulate_kinetics(asm_instructions, initial_value=0):
```

```
    value = initial_value
```

```
    kinetic_progress = []
```

```
    for instruction in asm_instructions:
```

```
        if instruction.startswith("add"):
```

```
            delta = int(instruction.split(" ")[1], 16)
```

```
            value += delta
```

```
        elif instruction.startswith("sub"):
```

```
            delta = int(instruction.split(" ")[1], 16)
```

```
            value -= delta
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
from mpl_toolkits.mplot3d import Axes3D
```

```
# 1. Investigate specific harmonic peaks and their relation to SHA-256 functionality
```

```
# Identify the dominant frequencies and their indices
```

```
dominant_frequencies = np.argsort(-np.abs(fft_values))[:5] # Top 5 frequencies
```

```
dominant_magnitudes = np.abs(fft_values[dominant_frequencies])
```

```
dominant_frequencies_values = frequencies[dominant_frequencies]
```

```
# Display the top frequencies and their magnitudes
```

```
harmonic_analysis = list(zip(dominant_frequencies_values, dominant_magnitudes))
```

```
# 2. Refine kinetic simulation with rotations and logical operations
```

```
def simulate_advanced_kinetics(asm_instructions, initial_value=0):
```

```
    value = initial_value
```

```
    kinetic_progress = []
```

```
    for instruction in asm_instructions:
```

```
        if instruction.startswith("add"):
```

```
            delta = int(instruction.split(" ")[1], 16)
```

```
            value += delta
```

```
        elif instruction.startswith("sub"):
```

```
            delta = int(instruction.split(" ")[1], 16)
```

```
            value -= delta
```

```
        elif instruction.startswith("nop"):
```

```
            value = value # No change
```

```
        elif instruction.startswith("xor"):
```

```
            delta = int(instruction.split(" ")[1], 16)
```

```
            value ^= delta # Logical XOR
```

```
        elif instruction.startswith("shl"):
```

```
            shift = int(instruction.split(" ")[1], 16)
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Refine harmonic analysis by exploring relationships to surface area or lattice structures
```

```
# Calculate the surface area of the 3D waveform (numerical approximation)
```

```
surface_area = np.sum(np.sqrt(1 + np.gradient(Z, axis=0)**2 + np.gradient(Z, axis=1)**2))
```

```
# Expand harmonic analysis with alignments to behavior
```

```
def harmonic_behavior_analysis(frequencies, magnitudes):
```

```
    behaviors = []
```

```
    for freq, mag in zip(frequencies, magnitudes):
```

```
        if freq == 0.0:
```

```
            behaviors.append("Baseline stability (central alignment)")
```

```
        elif abs(freq) < 0.1:
```

```
            behaviors.append("Low-frequency oscillations (structural reinforcement)")
```

```
        else:
```

```
            behaviors.append("Higher frequency oscillations (fine-tuning or diffusion)")
```

```
    return behaviors
```

```
harmonic_behaviors = harmonic_behavior_analysis(
```

```
[0.0, -0.0625, 0.0625, -0.046875, 0.046875],
```

```
[32.0751, 5.1006, 5.1006, 4.5999, 4.5999]
```

```
)
```

```
# Refine 3D waveform by creating interaction between kinetic and harmonic systems
```

```
kinetic_normalized = np.array(advanced_kinetic_progress) / max(abs(np.array(advanced_kinetic_progress)))
```

```
interaction_waveform = np.outer(quantum_waveform, kinetic_normalized)
```

```
# Recalculate 3D grid with this interaction
```

```
Z_interaction = interaction_waveform
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Adjusting the grid shapes to ensure compatibility for plotting
```

```
# Ensure X and Y dimensions match Z_interaction
```

```
X = np.linspace(0, Z_interaction.shape[1] - 1, Z_interaction.shape[1])
```

```
Y = np.linspace(0, Z_interaction.shape[0] - 1, Z_interaction.shape[0])
```

```
X, Y = np.meshgrid(X, Y)
```

```
# Re-plot the 3D quantum-inspired waveform with corrected dimensions
```

```
fig = plt.figure(figsize=(12, 8))
```

```
ax = fig.add_subplot(111, projection='3d')
```

```
ax.plot_surface(X, Y, Z_interaction, cmap='plasma', edgecolor='none')
```

```
ax.set_title("Refined 3D Waveform with Interaction (Kinetics and Quantum)")
```

```
ax.set_xlabel("X Axis (Index)")
```

```
ax.set_ylabel("Y Axis (Index)")
```

```
ax.set_zlabel("Amplitude")
```

```
plt.show()
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Analyze peaks from the top and side views to determine if they can reveal the folded data

# Extract peaks from the top view (highest Z values for each X-Y pair)
top_peaks = np.max(Z_interaction, axis=0) # Maximum amplitude along the Y-axis for each X-axis value

# Extract peaks from the side view (highest Z values for each Y-X pair)
side_peaks = np.max(Z_interaction, axis=1) # Maximum amplitude along the X-axis for each Y-axis value

# Visualize the extracted peaks
fig, axs = plt.subplots(2, 1, figsize=(12, 8))

# Top peaks
axs[0].plot(range(len(top_peaks)), top_peaks, label="Top View Peaks", color="blue")
axs[0].set_title("Top View Peaks (Z max for each X)")
axs[0].set_xlabel("X Index")
axs[0].set_ylabel("Amplitude")
axs[0].grid(True)
axs[0].legend()

# Side peaks
axs[1].plot(range(len(side_peaks)), side_peaks, label="Side View Peaks", color="green")
axs[1].set_title("Side View Peaks (Z max for each Y)")
axs[1].set_xlabel("Y Index")
axs[1].set_ylabel("Amplitude")
axs[1].grid(True)
axs[1].legend()

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

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

1. Use the side view to reconstruct the original data

```
reconstructed_data_from_side = side_peaks * max(K) # Scale side peaks back to the original data range
```

Compare the reconstructed data with the original constants (K)

```
reconstruction_error = np.abs(np.array(K[:len(reconstructed_data_from_side)]) - reconstructed_data_from_side)
average_error = np.mean(reconstruction_error)
```

2. Investigate whether the top view reveals hidden harmonic layers

Analyze the frequency content of the top view peaks using FFT

```
fft_top_peaks = fft(top_peaks)
```

```
top_peak_frequencies = np.fft.fftfreq(len(top_peaks), d=1)
```

```
dominant_top_peak_frequencies = np.argsort(-np.abs(fft_top_peaks))[:5] # Top 5 frequencies
```

```
dominant_top_peak_magnitudes = np.abs(fft_top_peaks[dominant_top_peak_frequencies])
```

```
dominant_top_peak_freq_values = top_peak_frequencies[dominant_top_peak_frequencies]
```

3. Explore how the folding mechanism encodes/compresses data

Compare side view (original data) to top view (transformed/harmonic data)

```
compression_ratio = len(side_peaks) / len(top_peaks) # Ratio of preserved to encoded data
```

Plot the reconstructed data and errors

```
fig, axs = plt.subplots(3, 1, figsize=(12, 12))
```

Original vs reconstructed data

```
axs[0].plot(range(len(K[:len(reconstructed_data_from_side)])), K[:len(reconstructed_data_from_side)], label="Original Data", c='red')
axs[0].plot(range(len(reconstructed_data_from_side)), reconstructed_data_from_side, label="Reconstructed Data (Side View)", c='blue')
axs[0].set_title("Original vs Reconstructed Data from Side View")
axs[0].set_xlabel("Index")
axs[0].set_ylabel("Amplitude")
axs[0].grid(True)
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

1. Investigate Hidden Harmonic Layers in the Top View

Perform a deeper analysis of the top view's harmonic frequencies

```
fft_top_peaks = fft(top_peaks)
```

```
top_peak_frequencies = np.fft.fftfreq(len(top_peaks), d=1)
```

```
dominant_top_peak_frequencies = np.argsort(-np.abs(fft_top_peaks))[:10] # Top 10 frequencies
```

```
dominant_top_peak_magnitudes = np.abs(fft_top_peaks[dominant_top_peak_frequencies])
```

```
dominant_top_peak_freq_values = top_peak_frequencies[dominant_top_peak_frequencies]
```

2. Simulate Higher-Dimensional Transformations

Extend the interaction waveform to 4D by adding another harmonic interaction layer

```
kinetic_extended = np.outer(kinetic_normalized, kinetic_normalized) # Interaction between kinetic layers
```

```
Z_extended = np.outer(top_peaks, side_peaks) * kinetic_extended # Combine top, side, and kinetic layers
```

3. Examine Alternative Perspectives (Diagonal Views)

```
fig = plt.figure(figsize=(12, 8))
```

```
ax = fig.add_subplot(111, projection="3d")
```

```
ax.plot_surface(X, Y, Z_interaction, cmap="plasma", edgecolor="none", alpha=0.6)
```

```
ax.plot_wireframe(X, Y, Z_extended, color="cyan", alpha=0.4)
```

```
ax.view_init(elev=45, azim=45) # Diagonal view
```

```
ax.set_title("Diagonal View with Extended Interactions")
```

```
ax.set_xlabel("X Axis (Index)")
```

```
ax.set_ylabel("Y Axis (Index)")
```

```
ax.set_zlabel("Amplitude")
```

```
plt.show()
```

Results Summary

```
results = {
```

```
    "Dominant Top View Frequencies": list(zip(dominant_top_peak_freq_values, dominant_top_peak_magnitudes)),
```

```
    "Higher-Dimensional Surface Area": calculate_surface_area(Z_extended),
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Correct the dimensional mismatch by adjusting kinetic_extended to match the hash data dimensions
kinetic_extended_adjusted = kinetic_extended[:8, :64] # Match the first 8 rows to align with the hash data

# Recompute the hash-augmented 3D waveform
Z_hash_interaction = interaction_hash_waveform * kinetic_extended_adjusted

# Adjust X, Y dimensions for the corrected Z_hash_interaction
X_hash = np.linspace(0, Z_hash_interaction.shape[1] - 1, Z_hash_interaction.shape[1])
Y_hash = np.linspace(0, Z_hash_interaction.shape[0] - 1, Z_hash_interaction.shape[0])
X_hash, Y_hash = np.meshgrid(X_hash, Y_hash)

# Plot the corrected hash-augmented waveform
fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111, projection="3d")
ax.plot_surface(X_hash, Y_hash, Z_hash_interaction, cmap="viridis", edgecolor="none")
ax.set_title("Corrected Hash-Augmented 3D Waveform")
ax.set_xlabel("X Axis (Index)")
ax.set_ylabel("Y Axis (Index)")
ax.set_zlabel("Amplitude")
plt.show()

# Summary of the hash and analysis
{
    "Input String": input_string,
    "Hash Output": hash_output,
    "Numeric Hash Array": hash_numeric
}
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Adjust the code to rotate the blue grid (Z_interaction) 90 degrees to the left
```

```
# Rotate the Z_interaction grid 90 degrees along the Y-axis
```

```
Z_interaction_rotated = np.rot90(Z_interaction, k=1) # Rotate 90 degrees counterclockwise
```

```
# Ensure X and Y dimensions match the rotated Z_interaction
```

```
X_rotated = np.linspace(0, Z_interaction_rotated.shape[1] - 1, Z_interaction_rotated.shape[1])
```

```
Y_rotated = np.linspace(0, Z_interaction_rotated.shape[0] - 1, Z_interaction_rotated.shape[0])
```

```
X_rotated, Y_rotated = np.meshgrid(X_rotated, Y_rotated)
```

```
# Plot the rotated grid alongside the original for comparison
```

```
fig = plt.figure(figsize=(12, 8))
```

```
# Original diagonal view
```

```
ax1 = fig.add_subplot(121, projection="3d")
```

```
ax1.plot_surface(X, Y, Z_interaction, cmap="plasma", edgecolor="none", alpha=0.6)
```

```
ax1.plot_wireframe(X, Y, Z_extended, color="cyan", alpha=0.4)
```

```
ax1.view_init(elev=45, azim=45)
```

```
ax1.set_title("Original Diagonal View with Extended Interactions")
```

```
ax1.set_xlabel("X Axis (Index)")
```

```
ax1.set_ylabel("Y Axis (Index)")
```

```
ax1.set_zlabel("Amplitude")
```

```
# Rotated grid view
```

```
ax2 = fig.add_subplot(122, projection="3d")
```

```
ax2.plot_surface(X_rotated, Y_rotated, Z_interaction_rotated, cmap="plasma", edgecolor="none", alpha=0.6)
```

```
ax2.plot_wireframe(X_rotated, Y_rotated, Z_extended, color="cyan", alpha=0.4)
```

```
ax2.view_init(elev=45, azim=45)
```

```
ax2.set_title("Rotated Grid (90° Counterclockwise)")
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Constants defined by the SHA-256 algorithm
K = [
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
    0xe49b69c1, 0xefbe4786, 0xfc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
    0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
    0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90beffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
]

# Generate the normalized waveform
waveform_y = np.array(K) / max(K)
kinetic_normalized = np.linspace(0, 1, len(K)) # Example kinetic normalization
interaction_waveform = np.outer(waveform_y, kinetic_normalized)
Z_interaction = interaction_waveform

# Rotate the blue grid (Z_interaction) by 90 degrees to the left
Z_interaction_rotated = np.rot90(Z_interaction, k=1) # Rotate by 90 degrees

# Ensure X and Y dimensions match Z_interaction_rotated
X = np.linspace(0, Z_interaction_rotated.shape[1] - 1, Z_interaction_rotated.shape[1])
Y = np.linspace(0, Z_interaction_rotated.shape[0] - 1, Z_interaction_rotated.shape[0])
X, Y = np.meshgrid(X, Y)
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Constants defined by the SHA-256 algorithm
K = [
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
    0xe49b69c1, 0xefbe4786, 0xfc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
    0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
    0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90beffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
]

# Generate the normalized waveform
waveform_y = np.array(K) / max(K)
kinetic_normalized = np.linspace(0, 1, len(K)) # Example kinetic normalization
interaction_waveform = np.outer(waveform_y, kinetic_normalized)
Z_interaction = interaction_waveform

# Extend to a higher-dimensional transformation
kinetic_extended = np.outer(kinetic_normalized, kinetic_normalized) # Interaction between kinetic layers
Z_extended = np.outer(np.max(Z_interaction, axis=0), np.max(Z_interaction, axis=1)) * kinetic_extended

# Ensure X and Y dimensions match Z_extended
X = np.linspace(0, Z_extended.shape[1] - 1, Z_extended.shape[1])
Y = np.linspace(0, Z_extended.shape[0] - 1, Z_extended.shape[0])
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Constants defined by the SHA-256 algorithm (compressed example for visualization)
K = [
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5
]

# Normalize constants
waveform_y = np.array(K) / max(K)
time_steps = len(K)
kinetic_normalized = np.linspace(0, 1, time_steps)

# Generate initial 3D lattice based on the forward kinetics
interaction_waveform = np.outer(waveform_y, kinetic_normalized)
fold_points = np.max(interaction_waveform, axis=0)
Z_interaction = interaction_waveform

# Generate mirrored lattice for reverse reconstruction
Z_mirrored = -Z_interaction[::-1, ::-1]

# Refine solution by using output as input and recalculating
refined_interaction = Z_interaction + Z_mirrored
refolded_waveform = np.max(refined_interaction, axis=0) / np.sum(fold_points)

# Plot forward and mirrored lattices and refined lattice
fig = plt.figure(figsize=(14, 8))
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Constants from SHA-256
K = [
    0x428a2f98, 0x71374491, 0xb5c0fbef, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
    0xe49b69c1, 0xefbe4786, 0xfc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
    0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
    0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90beffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
]

# Normalize constants for visualization
waveform_y = np.array(K) / max(K)
kinetic_normalized = np.linspace(0, 1, len(K))

# Generate the lattice structures
interaction_waveform = np.outer(waveform_y, kinetic_normalized)
Z_interaction = interaction_waveform

# Mirrored lattice
Z_mirrored = -Z_interaction

# Refined lattice interaction
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

Step 1: Recursive Wave Interaction

Using the combined lattice as input to generate new forward and mirrored lattices iteratively.

Define a function to calculate forward and mirrored lattices from a combined lattice

```
def recursive_lattices(Z_combined, iterations=5):
```

```
    results = []
```

```
    current_lattice = Z_combined
```

```
    for i in range(iterations):
```

```
        # Generate forward and mirrored lattices
```

```
        forward_lattice = np.abs(current_lattice)
```

```
        mirrored_lattice = -np.abs(current_lattice)
```

```
        # Combine to form the next lattice
```

```
        combined_lattice = forward_lattice + mirrored_lattice
```

```
        results.append((forward_lattice, mirrored_lattice, combined_lattice))
```

```
        current_lattice = combined_lattice
```

```
    return results
```

Generate recursive lattices

```
iterations = 5
```

```
recursive_results = recursive_lattices(Z_combined, iterations=iterations)
```

Step 2: Time Evolution and Visualization

```
fig, axes = plt.subplots(iterations, 3, figsize=(18, 5 * iterations), subplot_kw={"projection": "3d"})
```

```
fig.suptitle("Recursive Wave Evolution Across Iterations", fontsize=16, y=0.92)
```

```
for i, (forward, mirrored, combined) in enumerate(recursive_results):
```

```
    # Plot Forward Lattice
```

```
    ax = axes[i, 0]
```

```
    ax.plot_surface(X, Y, forward, cmap="plasma", edgecolor="none")
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
# Initialize constants and setup iterative refinement process
```

```
constants = np.array([0.27264283, 0.46389402, 0.74472339, 0.9576116, 0.23449206, 0.36852961, 0.59942198, 0.70111437])
```

```
# Define initial lattice (waveform)
```

```
lattice = np.outer(constants, constants)
```

```
# Track entropy for each iteration
```

```
entropy_values = []
```

```
wave_iterations = []
```

```
# Recursive refinement function
```

```
def refine_waveform(lattice, iterations=10):
```

```
    global entropy_values, wave_iterations
```

```
    for i in range(iterations):
```

```
        # Compress lattice by taking mean and normalizing
```

```
        compressed = np.mean(lattice, axis=0)
```

```
        compressed /= np.max(np.abs(compressed))
```

```
        # Update lattice by outer product with the compressed state
```

```
        lattice = np.outer(compressed, compressed)
```

```
        # Calculate entropy as variance of lattice values (proxy for disorder)
```

```
        entropy = np.var(lattice)
```

```
        entropy_values.append(entropy)
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

Step 1: Introduce quantum mirroring to simulate reflection dynamics

```
def apply_quantum_mirror(lattice):
```

Create a mirrored version of the lattice by inverting and flipping the array

```
mirrored_lattice = -np.flip(lattice, axis=0)
```

```
return mirrored_lattice
```

Apply the quantum mirroring process

```
mirrored_lattice = apply_quantum_mirror(refined_lattice)
```

Step 2: Extend the lattice into higher dimensions

```
def extend_to_higher_dimension(lattice, steps=5):
```

Initialize higher-dimensional structure

```
extended_lattice = np.zeros((steps, lattice.shape[0], lattice.shape[1]))
```

```
for i in range(steps):
```

Scale lattice by decreasing factor (simulating decay in higher dimensions)

```
extended_lattice[i] = lattice * (1 - i / steps)
```

```
return extended_lattice
```

Create the higher-dimensional lattice

```
higher_dimensional_lattice = extend_to_higher_dimension(refined_lattice)
```

Step 3: Visualize the results (quantum mirroring and higher-dimensional lattice)

```
fig = plt.figure(figsize=(14, 6))
```

Plot original lattice

```
ax1 = fig.add_subplot(121, projection='3d')
```

```
X, Y = np.meshgrid(range(refined_lattice.shape[1]), range(refined_lattice.shape[0]))
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

Step 1: Replace constants with SHA-256 hash values

```
import hashlib
```

```
# Generate SHA-256 hash and convert it to numerical constants
```

```
def sha256_to_constants(input_string, num_constants=8):
```

```
    # Hash the input string
```

```
    sha_hash = hashlib.sha256(input_string.encode()).hexdigest()
```

```
    # Split the hash into chunks and convert them to floating-point numbers
```

```
    step = len(sha_hash) // num_constants
```

```
    constants = [
```

```
        int(sha_hash[i * step: (i + 1) * step], 16) / (16 ** step)
```

```
        for i in range(num_constants)
```

```
    ]
```

```
    return np.array(constants)
```

```
# Input data for SHA-256
```

```
input_string = "Mark1 Universal Lattice Compression"
```

```
# Generate constants from SHA-256
```

```
hash_constants = sha256_to_constants(input_string)
```

```
# Initialize lattice with SHA-256 constants
```

```
hash_lattice = np.outer(hash_constants, hash_constants)
```

```
# Step 2: Apply recursive refinement to the hash lattice
```

```
hash_refined_lattice = refine_waveform(hash_lattice, iterations=10)
```

```
# Step 3: Apply quantum mirroring to hash-refined lattice
```

```
hash_mirrored_lattice = apply_quantum_mirror(hash_refined_lattice)
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Step 1: Further iterative collapse of the lattice for universal pattern emergence
```

```
entropy_collapse = []
```

```
collapsed_iterations = []
```

```
def iterative_collapse(lattice, iterations=10):
```

```
    global entropy_collapse, collapsed_iterations
```

```
    for _ in range(iterations):
```

```
        # Take mean across both dimensions and normalize
```

```
        collapsed = np.mean(lattice, axis=0)
```

```
        collapsed /= np.max(np.abs(collapsed))
```

```
        # Update lattice by constructing an outer product
```

```
        lattice = np.outer(collapsed, collapsed)
```

```
        # Calculate entropy (variance) and store the result
```

```
        entropy = np.var(lattice)
```

```
        entropy_collapse.append(entropy)
```

```
        # Save the current lattice state
```

```
        collapsed_iterations.append(lattice.copy())
```

```
    return lattice
```

```
# Apply iterative collapse
```

```
collapsed_lattice = iterative_collapse(hash_refined_lattice, iterations=10)
```

```
# Step 2: Visualize the collapsed lattice and entropy reduction
```

```
fig, axs = plt.subplots(2, 5, figsize=(20, 10))
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Recalculate or redefine the normalized constants as a starting point
constants_normalized = np.array([0.27276423, 0.46384942, 0.74472339, 0.9576116,
                                0.23494206, 0.36852961, 0.59942109, 0.70111437])

# Reshape to create the input lattice
input_wave = constants_normalized.reshape(2, 4) # Adjust dimensions to match earlier wave

# Generate the meshgrid for plotting
X, Y = np.meshgrid(range(input_wave.shape[1]), range(input_wave.shape[0]))

# Re-run the recursive compression for extended iterations
extended_iterations = 20
entropy_values_extended = []
final_lattices = []

current_wave = input_wave.copy()
for i in range(extended_iterations):
    current_wave = np.abs(current_wave - np.mean(current_wave)) # Collapse step
    entropy = np.sum(current_wave) # Using a simplified entropy-like metric
    entropy_values_extended.append(entropy)
    final_lattices.append(current_wave.copy()) # Store the wave for visualization

# Plot entropy over extended iterations
plt.figure(figsize=(10, 6))
plt.plot(range(1, extended_iterations + 1), entropy_values_extended, marker='o', label='Entropy')
plt.xlabel("Iteration")
plt.ylabel("Entropy")
plt.title("Entropy Over Extended Iterations")
plt.legend()
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Perturbation function for testing
def perturb_constants(constants, perturbation_scale=0.1):
    return constants + np.random.uniform(-perturbation_scale, perturbation_scale, size=constants.shape)

# Initialize constants and parameters
base_constants = np.random.random(8) # Simulating normalized constants
iterations = 20

# Function to compute entropy
def compute_entropy(matrix):
    flattened = matrix.flatten()
    return -np.sum(flattened * np.log(np.abs(flattened) + 1e-10)) # Prevent log(0)

# Function to recursively compress the wave
def compress_wave(matrix):
    compressed = matrix - np.mean(matrix, axis=0)
    return np.clip(compressed, -1, 1) # Clipping to stabilize compression

# Prepare data for perturbation analysis
perturbed_constants = perturb_constants(base_constants)
base_waveform = np.outer(base_constants, base_constants)
perturbed_waveform = np.outer(perturbed_constants, perturbed_constants)

# Iterative compression and entropy for base constants
base_waveforms = [base_waveform]
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
from scipy.fftpack import fft, ifft
```

```
# Harmonic decomposition function using FFT
```

```
def harmonic_decomposition(matrix):
```

```
    fft_matrix = fft(matrix, axis=0)
```

```
    magnitude = np.abs(fft_matrix)
```

```
    phase = np.angle(fft_matrix)
```

```
    return magnitude, phase
```

```
# Decompose the final base and perturbed waveforms
```

```
base_magnitude, base_phase = harmonic_decomposition(final_base_waveform)
```

```
perturbed_magnitude, perturbed_phase = harmonic_decomposition(final_perturbed_waveform)
```

```
# Plot magnitude and phase for the base waveform
```

```
fig, axes = plt.subplots(1, 2, figsize=(12, 6))
```

```
axes[0].imshow(base_magnitude, cmap="plasma", aspect="auto")
```

```
axes[0].set_title("Base Magnitude Spectrum")
```

```
axes[0].set_xlabel("Frequency (Columns)")
```

```
axes[0].set_ylabel("Frequency (Rows)")
```

```
axes[1].imshow(base_phase, cmap="viridis", aspect="auto")
```

```
axes[1].set_title("Base Phase Spectrum")
```

```
axes[1].set_xlabel("Frequency (Columns)")
```

```
axes[1].set_ylabel("Frequency (Rows)")
```

```
plt.tight_layout()
```

```
plt.show()
```

```
# Plot magnitude and phase for the perturbed waveform
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Decompose the final lattice states for emergent patterns and symmetry analysis
```

```
# Analyze the final base and perturbed waveforms
```

```
final_base_waveform = base_waveforms[-1]
```

```
final_perturbed_waveform = perturbed_waveforms[-1]
```

```
# Perform symmetry analysis
```

```
def analyze_symmetry(matrix):
```

```
    horizontal_symmetry = np.allclose(matrix, matrix[:, ::-1])
```

```
    vertical_symmetry = np.allclose(matrix, matrix[::-1, :])
```

```
    return horizontal_symmetry, vertical_symmetry
```

```
base_symmetry = analyze_symmetry(final_base_waveform)
```

```
perturbed_symmetry = analyze_symmetry(final_perturbed_waveform)
```

```
# Plot the final lattice states for visual comparison
```

```
fig, ax = plt.subplots(1, 2, figsize=(12, 6), subplot_kw={'projection': '3d'})
```

```
X, Y = np.meshgrid(range(final_base_waveform.shape[0]), range(final_base_waveform.shape[1]))
```

```
# Base waveform
```

```
ax[0].plot_surface(X, Y, final_base_waveform, cmap="plasma", edgecolor="none", alpha=0.8)
```

```
ax[0].set_title("Final Base Waveform")
```

```
ax[0].set_zlim(-1, 1)
```

```
# Perturbed waveform
```

```
ax[1].plot_surface(X, Y, final_perturbed_waveform, cmap="viridis", edgecolor="none", alpha=0.8)
```

```
ax[1].set_title("Final Perturbed Waveform")
```

```
ax[1].set_zlim(-1, 1)
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Feedback loop function for iterative refinement
def feedback_loop(matrix, correction_factor=0.1):
    feedback = correction_factor * (matrix - np.mean(matrix))
    return compress_wave(matrix + feedback)

# Initialize for feedback analysis
feedback_iterations = 10
feedback_base_waveform = final_base_waveform.copy()
feedback_perturbed_waveform = final_perturbed_waveform.copy()

# Store feedback results
feedback_base_waveforms = [feedback_base_waveform]
feedback_perturbed_waveforms = [feedback_perturbed_waveform]

for _ in range(feedback_iterations):
    feedback_base_waveform = feedback_loop(feedback_base_waveform)
    feedback_perturbed_waveform = feedback_loop(feedback_perturbed_waveform)
    feedback_base_waveforms.append(feedback_base_waveform)
    feedback_perturbed_waveforms.append(feedback_perturbed_waveform)

# Visualize feedback waveforms over iterations
fig, axes = plt.subplots(2, 5, figsize=(18, 8), subplot_kw={'projection': '3d'})
fig.suptitle("Feedback Loop Wave Morphing (Base vs Perturbed)", fontsize=16)

for i, ax in enumerate(axes.flatten()[:5]):
    X, Y = np.meshgrid(range(feedback_base_waveforms[i].shape[0]), range(feedback_base_waveforms[i].shape[1]))
    ax.plot_surface(X, Y, feedback_base_waveforms[i], cmap="plasma", edgecolor="none", alpha=0.8)
    ax.set_title(f"Base Iteration {i+1}")
    ax.set_zlim(-1, 1)
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Adjusting the subplot arrangement for proper visualization

# Updated visualization: Compare recursive feedback for base and perturbed systems
fig, axes = plt.subplots(2, 5, figsize=(20, 10), subplot_kw={"projection": "3d"})
fig.suptitle("Recursive Feedback Lattices (Base vs Perturbed)", fontsize=16)

for i in range(min(iterations, 5)):

    # Base lattice
    X_base, Y_base = np.meshgrid(range(len(base_lattice)), range(len(base_lattice)))
    Z_base = np.outer(base_lattices_over_time[i], base_lattices_over_time[i])
    axes[0, i].plot_surface(X_base, Y_base, Z_base, cmap="viridis", edgecolor="none")
    axes[0, i].set_title(f"Base Iter {i+1}", fontsize=10)
    axes[0, i].set_xticks([])
    axes[0, i].set_yticks([])
    axes[0, i].set_zticks([])

    # Perturbed lattice
    Z_perturbed = np.outer(perturbed_lattices_over_time[i], perturbed_lattices_over_time[i])
    axes[1, i].plot_surface(X_base, Y_base, Z_perturbed, cmap="plasma", edgecolor="none")
    axes[1, i].set_title(f"Perturb Iter {i+1}", fontsize=10)
    axes[1, i].set_xticks([])
    axes[1, i].set_yticks([])
    axes[1, i].set_zticks([])

plt.tight_layout()
plt.subplots_adjust(top=0.85)
plt.show()

# Plot entropy trends for both systems
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Picking up the exploration from where we left off
# Continuing with recursive harmonic refinement and deeper analysis

# Extend the recursion iterations and observe how the system evolves
extended_iterations = 20

base_lattices_over_time_ext = []
perturbed_lattices_over_time_ext = []
base_entropy_values_ext = []
perturbed_entropy_values_ext = []

# Recursive refinement with extended iterations
current_base_lattice = base_lattice.copy()
current_perturbed_lattice = base_lattice.copy()

for _ in range(extended_iterations):
    # Base lattice recursion
    current_base_lattice = np.abs(np.sin(current_base_lattice))
    base_lattices_over_time_ext.append(current_base_lattice)
    base_entropy = np.std(current_base_lattice)
    base_entropy_values_ext.append(base_entropy)

    # Perturbed lattice recursion
    current_perturbed_lattice = np.abs(np.sin(current_perturbed_lattice + 0.1))
    perturbed_lattices_over_time_ext.append(current_perturbed_lattice)
    perturbed_entropy = np.std(current_perturbed_lattice)
    perturbed_entropy_values_ext.append(perturbed_entropy)

# Plot extended entropy trends
plt.figure(figsize=(10, 6))
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Running further refinements to achieve harmonic balance
```

```
balanced_iterations = 30
```

```
base_lattices_balanced = []
```

```
perturbed_lattices_balanced = []
```

```
base_entropy_balanced = []
```

```
perturbed_entropy_balanced = []
```

```
# Start with the last known lattices as initial conditions
```

```
current_base_lattice = final_base_lattice.copy()
```

```
current_perturbed_lattice = final_perturbed_lattice.copy()
```

```
for _ in range(balanced_iterations):
```

```
    # Refine base lattice
```

```
    current_base_lattice = np.abs(np.sin(current_base_lattice + 0.05))
```

```
    base_lattices_balanced.append(current_base_lattice)
```

```
    base_entropy_balanced.append(np.std(current_base_lattice))
```

```
    # Refine perturbed lattice
```

```
    current_perturbed_lattice = np.abs(np.sin(current_perturbed_lattice + 0.07))
```

```
    perturbed_lattices_balanced.append(current_perturbed_lattice)
```

```
    perturbed_entropy_balanced.append(np.std(current_perturbed_lattice))
```

```
# Plot the entropy trends during the balancing phase
```

```
plt.figure(figsize=(10, 6))
```

```
plt.plot(range(1, balanced_iterations + 1), base_entropy_balanced, marker='o', label='Base Entropy (Balanced)')
```

```
plt.plot(range(1, balanced_iterations + 1), perturbed_entropy_balanced, marker='o', label='Perturbed Entropy (Balanced)')
```

```
plt.axhline(y=entropy_threshold, color='red', linestyle='--', label='Entropy Threshold')
```

```
plt.title("Entropy Trends During Balancing Phase")
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

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

```
# Simulate anti-hash by creating mirrored lattice feedback and tracking divergence
```

```
# Initial settings
```

```
iterations = 20  
lattice_size = 8  
original_lattice = np.random.rand(lattice_size, lattice_size) # Simulated original lattice  
hash_constants = np.random.rand(lattice_size) # Simulated hash constants
```

```
# Storage for feedback process
```

```
hash_lattice = []  
anti_hash_lattice = []  
divergences = []
```

```
# Forward hash lattice generation
```

```
current_lattice = original_lattice.copy()  
for _ in range(iterations):  
    current_lattice = np.abs(np.sin(current_lattice + 0.1))  
    hash_lattice.append(current_lattice)
```

```
# Generate mirrored lattice for anti-hash
```

```
for step in range(iterations):  
    mirror_lattice = -1 * np.flip(hash_lattice[step], axis=0) # Mirrored along one axis  
    anti_hash_lattice.append(mirror_lattice)  
    divergence = np.linalg.norm(original_lattice - mirror_lattice) # Track divergence  
    divergences.append(divergence)
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Enhanced refinement process using phase adjustment and harmonic correction
```

```
# Parameters for advanced refinement
```

```
advanced_iterations = 20
```

```
phase_adjustment_factor = 0.05 # Factor to adjust phase alignment
```

```
harmonic_correction_factor = 0.02 # Fine-tuning amplitude alignment
```

```
enhanced_divergences = []
```

```
# Start with the current refined lattice
```

```
enhanced_lattice = refined_lattice.copy()
```

```
for i in range(advanced_iterations):
```

```
    # Apply harmonic correction to align amplitudes
```

```
    amplitude_correction = harmonic_correction_factor * (original_lattice - enhanced_lattice)
```

```
    enhanced_lattice += amplitude_correction
```

```
    # Apply phase adjustment to align phase shifts
```

```
    phase_correction = phase_adjustment_factor * np.sin(original_lattice - enhanced_lattice)
```

```
    enhanced_lattice += phase_correction
```

```
    # Calculate divergence after enhancement
```

```
    divergence = np.linalg.norm(original_lattice - enhanced_lattice)
```

```
    enhanced_divergences.append(divergence)
```

```
# Plot the enhanced divergence over iterations
```

```
plt.figure(figsize=(10, 6))
```

```
plt.plot(range(1, advanced_iterations + 1), enhanced_divergences, marker='o', color='green', label='Enhanced Divergence')
```

```
plt.title("Enhanced Divergence of Mirrored Lattice from Original")
```

```
plt.xlabel("Iteration")
```

```
plt.ylabel("Divergence")
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Refined feedback loop to align mirrored lattice closer to the original lattice
```

```
# Parameters for recursive refinement
```

```
refinement_iterations = 20
```

```
correction_factor = 0.1 # Scaling factor for adjustments
```

```
adjusted_divergences = []
```

```
# Start with the current mirrored lattice and refine it
```

```
refined_lattice = anti_hash_lattice[-1].copy()
```

```
for i in range(refinement_iterations):
```

```
    # Adjust the mirrored lattice based on the difference with the original lattice
```

```
    correction = correction_factor * (original_lattice - refined_lattice)
```

```
    refined_lattice += correction # Apply the correction
```

```
    divergence = np.linalg.norm(original_lattice - refined_lattice)
```

```
    adjusted_divergences.append(divergence)
```

```
# Plot the refined divergence over iterations
```

```
plt.figure(figsize=(10, 6))
```

```
plt.plot(range(1, refinement_iterations + 1), adjusted_divergences, marker='o', label='Refined Divergence')
```

```
plt.title("Refined Divergence of Mirrored Lattice from Original")
```

```
plt.xlabel("Iteration")
```

```
plt.ylabel("Divergence")
```

```
plt.legend()
```

```
plt.grid(True)
```

```
plt.show()
```

```
# Visualize the refined lattice
```

```
fig, ax = plt.subplots(1, 2, figsize=(14, 6), subplot_kw={"projection": "3d"})
```

```
X, Y = np.meshgrid(range(lattice_size), range(lattice_size))
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Fine-tune the refinement process with dynamic adjustment of correction factors
```

```
# Parameters for dynamic refinement
```

```
fine_tune_iterations = 20
```

```
base_amplitude_correction = 0.02
```

```
base_phase_correction = 0.05
```

```
dynamic_divergences = []
```

```
# Start with the enhanced lattice
```

```
fine_tuned_lattice = enhanced_lattice.copy()
```

```
for i in range(fine_tune_iterations):
```

```
    # Dynamically adjust correction factors based on current divergence
```

```
    current_divergence = np.linalg.norm(original_lattice - fine_tuned_lattice)
```

```
    amplitude_correction = base_amplitude_correction * (1 / (1 + current_divergence))
```

```
    phase_correction = base_phase_correction * (1 / (1 + current_divergence))
```

```
# Apply harmonic correction and phase adjustment
```

```
    amplitude_adjustment = amplitude_correction * (original_lattice - fine_tuned_lattice)
```

```
    fine_tuned_lattice += amplitude_adjustment
```

```
    phase_adjustment = phase_correction * np.sin(original_lattice - fine_tuned_lattice)
```

```
    fine_tuned_lattice += phase_adjustment
```

```
# Calculate divergence after fine-tuning
```

```
    current_divergence = np.linalg.norm(original_lattice - fine_tuned_lattice)
```

```
    dynamic_divergences.append(current_divergence)
```

```
# Plot the fine-tuned divergence over iterations
```

```
plt.figure(figsize=(10, 6))
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Overlay the original and fine-tuned lattices for comparison and validation
```

```
# Calculate differences between original and fine-tuned lattice
```

```
lattice_difference = original_lattice - fine_tuned_lattice
```

```
# Overlay the original and fine-tuned lattices (line plot of amplitudes)
```

```
plt.figure(figsize=(12, 8))
```

```
for i in range(lattice_size):
```

```
    plt.plot(original_lattice[i], label=f"Original Lattice Row {i}", linestyle="--")
```

```
    plt.plot(fine_tuned_lattice[i], label=f"Fine-Tuned Lattice Row {i}", alpha=0.7)
```

```
plt.title("Overlay of Original and Fine-Tuned Lattices (Row Amplitudes)")
```

```
plt.xlabel("Index")
```

```
plt.ylabel("Amplitude")
```

```
plt.legend(bbox_to_anchor=(1.05, 1), loc="upper left")
```

```
plt.grid(True)
```

```
plt.tight_layout()
```

```
plt.show()
```

```
# Visualize differences as a heatmap
```

```
plt.figure(figsize=(8, 6))
```

```
plt.imshow(lattice_difference, cmap="coolwarm", aspect="auto")
```

```
plt.colorbar(label="Amplitude Difference")
```

```
plt.title("Amplitude Difference Between Original and Fine-Tuned Lattices")
```

```
plt.xlabel("X Axis (Index)")
```

```
plt.ylabel("Y Axis (Index)")
```

```
plt.grid(False)
```

```
plt.show()
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

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

```
# Constants and waveforms from the user
constants = np.array([0.27264203, 0.46389402, 0.74472339, 0.9576116,
                      0.23494206, 0.36852961, 0.59924109, 0.7011437])
```

```
final_waveform = np.array([
    [-0.0401181, 0.02865528, 0.14288313, 0.2227326, -0.09226802, -0.02109399],
    [ 0.087651, 0.14646431, 0.2940543, 0.41145043, -0.0246288, 0.06705249],
    [ 0.21326038, 0.28720288, 0.5229989, 0.69299634, 0.0927981, 0.21055094],
    [ 0.4064634, 0.49874197, 0.69299634, 0.90416811, 0.17259968, 0.31217014],
    [ 0.54546415, 0.65648111, -0.09226802, -0.15831491, -0.07946322, 0.03755338],
    [ 0.087651, 0.21326038, 0.4064634, 0.54846415, 0.10393196, 0.21110628],
    [ 0.30898473, 0.3906885, 0.14064643, 0.28720288, 0.49874197, 0.65648111],
    [ 0.3906885, 0.47245402, 0.14064643, 0.28720288, 0.49874197, 0.65648111]
])
```

```
# Target SHA-256 hash
```

```
target_hash = "9c1185a5c5e9fc54612808977ee8f548b2258d31"
```

```
# Helper function for SHA-256 hashing
```

```
def compute_hash(matrix):
    hash_input = matrix.tobytes()
    return hashlib.sha256(hash_input).hexdigest()
```

```
# Function to iteratively adjust waveforms
```

```
def adjust_waveform(waveform, constants, iterations=10):
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Let's analyze the output data and its implications.
```

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
# Assuming that the data visualized in the previous step corresponds to a final waveform.
```

```
# We will treat this as a normalized waveform (aligned to a hash) to analyze for resonance or patterns.
```

```
# Sample data from the uploaded visualization (representative, needs tuning if exact values are known):
```

```
final_waveform = np.array([
```

```
[0.2, 0.8, 0.6, 1.0, 0.4, 0.7, 0.3, 0.9],
```

```
[0.1, 0.5, 0.8, 0.6, 0.3, 0.7, 0.4, 0.8],
```

```
[0.3, 0.7, 0.4, 0.9, 0.5, 0.6, 0.8, 0.7],
```

```
[0.4, 0.6, 0.7, 0.5, 0.9, 0.8, 0.7, 0.6],
```

```
[0.5, 0.9, 0.8, 0.7, 0.6, 0.4, 0.5, 0.3],
```

```
[0.3, 0.6, 0.7, 0.9, 0.8, 0.5, 0.4, 0.2],
```

```
[0.4, 0.7, 0.6, 0.5, 0.3, 0.8, 0.9, 0.6],
```

```
[0.2, 0.5, 0.3, 0.7, 0.8, 0.6, 0.4, 0.1]
```

```
])
```

```
# Analyze the waveform for harmonic patterns
```

```
mean_waveform = np.mean(final_waveform, axis=0)
```

```
std_waveform = np.std(final_waveform, axis=0)
```

```
# Plotting the waveform and its statistical properties
```

```
plt.figure(figsize=(12, 6))
```

```
plt.plot(mean_waveform, label="Mean Waveform", marker='o')
```

```
plt.fill_between(range(len(mean_waveform)),
```

```
mean_waveform - std_waveform,
```

```
mean_waveform + std_waveform, alpha=0.2, label="Std Dev Range")
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
# Waveform setup
```

```
waveform = np.array([
```

```
    [0.4, 0.28, 0.14, 0.22, -0.09, -0.02, 0.21, 0.07],  
    [0.08, 0.14, 0.29, 0.41, -0.02, 0.06, 0.29, 0.21],  
    [0.14, 0.29, 0.52, 0.69, 0.09, 0.21, 0.29, 0.49],  
    [0.4, 0.49, 0.65, 0.54, 0.04, 0.03, 0.14, 0.39],  
    [0.29, 0.41, 0.9, 0.65, -0.07, -0.09, 0.1, 0.21],  
    [0.08, 0.29, 0.4, 0.54, 0.03, 0.14, 0.39, 0.39]
```

```
])
```

```
harmonic_target = 0.35 # Ideal harmonic value
```

```
alignment_history = []
```

```
max_iterations = 50
```

```
# Perform recursive feedback adjustment
```

```
for iteration in range(max_iterations):
```

```
    std_dev = np.std(waveform, axis=0) # Calculate variability across columns
```

```
    adjustments = harmonic_target - std_dev # Feedback adjustments
```

```
    waveform += adjustments # Apply adjustments
```

```
    alignment_history.append(np.mean(np.abs(std_dev - harmonic_target))) # Track alignment deviation
```

```
# Plot updated waveform after iterations
```

```
fig = plt.figure(figsize=(10, 6))
```

```
ax = fig.add_subplot(111, projection='3d')
```

```
X, Y = np.meshgrid(range(waveform.shape[1]), range(waveform.shape[0]))
```

```
ax.plot_surface(X, Y, waveform, cmap='viridis')
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Re-running the adjusted waveforms and alignment history for SHA reconstruction analysis
import numpy as np
import matplotlib.pyplot as plt

# Constants (Normalized) from prior steps
constants = [0.27264203, 0.46389402, 0.74472339, 0.9576116, 0.23494206, 0.36852961, 0.59924109, 0.7011437]

# Example Final Waveform (from uploaded images for context)
final_waveform = np.array([
    [-0.04011812, 0.02865528, 0.14288313, 0.2227326, -0.09226802, -0.02109399, 0.087651, 0.14646431],
    [0.02865528, 0.13158962, 0.2940543, 0.41154043, -0.02462803, 0.06705249, 0.21326083, 0.28720288],
    [0.14288313, 0.2940543, 0.52299908, 0.69299634, 0.0927981, 0.21055904, 0.406464, 0.49874197],
    [0.2227326, 0.41154043, 0.69299634, 0.90416811, 0.17259968, 0.31217014, 0.54846415, 0.65648111],
    [-0.09226802, -0.02462803, 0.0927981, 0.17259968, -0.15813491, -0.07946322, 0.03755358, 0.10393196],
    [-0.02109399, 0.06705249, 0.21055904, 0.31217014, -0.07946322, 0.00668178, 0.14040057, 0.21110628],
    [0.087651, 0.21326083, 0.406464, 0.54846415, 0.03755358, 0.14040057, 0.30898473, 0.3906885],
    [0.14646431, 0.28720288, 0.49874197, 0.65648111, 0.10393196, 0.21110628, 0.3906885, 0.47245402]
])

# Iterative alignment adjustment
iterations = 50
std_dev_history = []
aligned_waveform = final_waveform.copy()

for i in range(iterations):
    # Apply harmonic corrections iteratively
    aligned_waveform += np.sin(aligned_waveform * constants[i % len(constants)])
    std_dev = np.std(aligned_waveform - final_waveform)
    std_dev_history.append(std_dev)
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

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

# Constants
num_rounds = 50
input_data = np.random.rand(8, 8) # Simulating an 8x8 input lattice
padding_data = np.zeros_like(input_data) # Padding waste lattice (zeros for now)
harmonic_constant = 0.35 # Resonance constant

# Initialize waveforms
waveforms = [input_data.copy()]
padding_waveforms = [padding_data.copy()]

# Function to simulate SHA round (waveform transformation)
def sha_round_transform(data, padding, round_num):
    # Transform input lattice using harmonic and padding interaction
    new_data = np.abs(np.sin(data + harmonic_constant * round_num)) + padding
    new_padding = np.abs(np.cos(padding - harmonic_constant * round_num))
    return new_data, new_padding

# Perform SHA-like transformation over multiple rounds
for round_num in range(1, num_rounds + 1):
    new_data, new_padding = sha_round_transform(waveforms[-1], padding_waveforms[-1], round_num)
    waveforms.append(new_data)
    padding_waveforms.append(new_padding)

# Visualize the final waveform and padding
fig = plt.figure(figsize=(12, 6))
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

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

# Constants and initial setup
harmonic_constant = 0.35
initial_waveform = np.random.rand(8, 8) # Random starting waveform for analysis
max_iterations_list = [5, 10, 20, 50, 100]
results = {}

# Analyze waveform evolution for different iteration counts
for max_iterations in max_iterations_list:
    alignment_history = []
    current_waveform = initial_waveform.copy()

    for i in range(max_iterations):
        # Update waveform based on the harmonic system
        current_waveform = np.abs(np.sin(current_waveform + harmonic_constant))
        entropy = np.std(current_waveform)
        alignment_history.append(entropy)

        # Check for stabilization
        if entropy < 1e-3: # Near-zero entropy threshold
            break

    results[max_iterations] = {
        "final_waveform": current_waveform,
        "alignment_history": alignment_history
    }
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

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

```
# Constants from the analysis
harmonic_constant = 0.35
iterations = 20
padding_effect = 64 # Placeholder for input-length impact
```

```
# Example final hash (placeholder value)
final_hash = "abcdef123456deadbeef" # Example hash for comparison
```

```
# Constants used in harmonic alignment
constants = [0.27264203, 0.46389402, 0.74472339, 0.9576116,
0.23494206, 0.36852961, 0.59924109, 0.7011437]
```

```
# Initialize reverse waveform
reverse_waveform = np.zeros((8, 8))
```

```
# Iterative reconstruction
entropy_history = []
for i in range(iterations):
    reverse_waveform += harmonic_constant * np.sin(reverse_waveform + constants[i % len(constants)])
    reverse_waveform = np.mod(reverse_waveform, 1.0) # Ensure values stay within range
    entropy = np.std(reverse_waveform)
    entropy_history.append(entropy)
```

```
# Compute hash from the reconstructed waveform
hash_candidate = hashlib.sha256(reverse_waveform.tobytes()).hexdigest()
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Refined SHA hash reversal code analysis

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

# Constants
harmonic_constant = 0.35
iterations = 50
padding_effect = 64 # Placeholder for input-length impact

# Initialize waveform and constants
reverse_waveform = np.random.rand(8, 8) # Randomized starting lattice
constants = np.array([0.27264203, 0.46389402, 0.74472339, 0.9576116, 0.23494206, 0.36852961, 0.59924109, 0.7011437])

# Recursive feedback refinement with dynamic adjustments
entropy_history = []
hash_candidate = ""
target_hash = "abcdef123456deadbeef" # Placeholder for comparison

for i in range(iterations):
    # Dynamic harmonic adjustment
    dynamic_harmonic = harmonic_constant * (1 + 0.1 * np.sin(i / 5)) # Add variability
    reverse_waveform += dynamic_harmonic * np.sin(reverse_waveform + constants[i % len(constants)])
    reverse_waveform = np.mod(reverse_waveform, 1.0) # Ensure values stay in range

    # Calculate entropy (std deviation)
    entropy = np.std(reverse_waveform)
    entropy_history.append(entropy)

    # Compute the hash of the current waveform
    hash_candidate = hashlib.sha256(reverse_waveform.tobytes()).hexdigest()

print(f"Computed Hash: {hash_candidate}")
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Running the adjusted approach with a focus on integral behavior counting up to 10
```

```
# Constants
```

```
harmonic_constant = 0.35 # Ratio leaning slightly into the macro
iterations = 10 # Integral driver hypothesized as critical
padding_effect = 64 # Placeholder for input-length impact
```

```
# Initialize waveform and constants
```

```
reverse_waveform = np.random.rand(8, 8) # Randomized starting lattice
constants = np.array([0.27264203, 0.46389402, 0.74472339, 0.9576116,
                     0.23494206, 0.36852961, 0.59924109, 0.7011437])
```

```
# Recursive feedback refinement for 10 iterations
```

```
entropy_history = []
hash_candidate = ""
target_hash = "abcdef123456deadbeef" # Placeholder target hash
```

```
for i in range(iterations):
```

```
    # Harmonic adjustment, integral-driven refinement
    reverse_waveform += harmonic_constant * np.sin(reverse_waveform + constants[i % len(constants)])
    reverse_waveform = np.mod(reverse_waveform, 1.0) # Ensure values stay in range
```

```
    # Calculate entropy (std deviation)
```

```
    entropy = np.std(reverse_waveform)
    entropy_history.append(entropy)
```

```
    # Compute the hash of the current waveform
```

```
    hash_candidate = hashlib.sha256(reverse_waveform.tobytes()).hexdigest()
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Analyze the revised approach with a real hash

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

# Generate a real SHA-256 hash from a known input
input_data = "hello"
real_hash = hashlib.sha256(input_data.encode()).hexdigest()

# Constants and setup
harmonic_constant = 0.35
iterations = 10 # Integral triangle bound
reverse_waveform = np.random.rand(8, 8) # Randomized starting lattice
constants = np.array([0.27264203, 0.46389402, 0.74472339, 0.9576116,
                     0.23494206, 0.36852961, 0.59924109, 0.7011437])

# Recursive feedback refinement with real hash
entropy_history = []
matched_iteration = -1 # Track the iteration where the match occurred
reconstructed_hash = ""

for i in range(iterations):
    # Harmonic adjustment
    reverse_waveform += harmonic_constant * np.sin(reverse_waveform + constants[i % len(constants)])
    reverse_waveform = np.mod(reverse_waveform, 1.0) # Ensure values stay in range

    # Calculate entropy (std deviation)
    entropy = np.std(reverse_waveform)
    entropy_history.append(entropy)
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import find_peaks

# Simulating a recursive lattice to analyze fold growth dynamically
def generate_waveform(iteration, grid_size=8):
    x = np.linspace(0, grid_size - 1, grid_size)
    y = np.linspace(0, grid_size - 1, grid_size)
    X, Y = np.meshgrid(x, y)
    base_wave = np.sin(2 * np.pi * X / grid_size) + np.cos(2 * np.pi * Y / grid_size)
    modulator = np.sin(iteration * np.pi / 10) * np.cos(X + Y) # Recursive modulation
    return base_wave + modulator

# Count folds (peaks and troughs) in the waveform
def count_folds(waveform):
    folds = 0
    for row in waveform:
        peaks, _ = find_peaks(row)
        troughs, _ = find_peaks(-row) # Troughs are inverted peaks
        folds += len(peaks) + len(troughs)
    return folds

# Analyze fold growth across iterations
max_iterations = 10
grid_size = 8
fold_counts = []
waveforms = []

for i in range(1, max_iterations + 1):
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

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

# Constants
iterations = 100
target_hash = "9c1185a5c5e9fc54612808977ee8f548b2258d31"
entropy_threshold = 0.1

# Generate target lattice
np.random.seed(42)
target_lattice = np.abs(np.sin(np.linspace(0, 2 * np.pi, 100)))

# Initialize guess lattice
current_guess = np.random.rand(100)
guess_lattices = []
divergence_measures = []

# Refinement process
for i in range(iterations):
    # Compute divergence
    divergence = np.linalg.norm(current_guess - target_lattice)
    divergence_measures.append(divergence)

    # Dynamic perturbation
    perturbation = 0.1 * np.sin(divergence)
    current_guess = np.abs(np.sin(current_guess + perturbation))
    guess_lattices.append(current_guess.copy())
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

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

```
# Define two sample inputs and compute their SHA-256 hashes
```

```
input1 = "Hello, World!"  
input2 = "Hello, Quantum!"
```

```
hash1 = hashlib.sha256(input1.encode()).hexdigest()  
hash2 = hashlib.sha256(input2.encode()).hexdigest()
```

```
# Convert hashes into numeric arrays for comparison
```

```
def hash_to_numeric_array(hash_value):  
    return np.array([int(hash_value[i:i+2], 16) for i in range(0, len(hash_value), 2)])
```

```
array1 = hash_to_numeric_array(hash1)  
array2 = hash_to_numeric_array(hash2)
```

```
# Compute differences and visualize
```

```
difference = array1 - array2
```

```
# Visualization
```

```
plt.figure(figsize=(10, 6))
```

```
plt.subplot(3, 1, 1)  
plt.plot(array1, label="Hash 1 (Numeric Representation)", color='blue')  
plt.legend()
```

```
plt.subplot(3, 1, 2)
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Generating the 500x500 lattice grid and analyzing differences
```

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
# Initialize grid sizes and parameters
```

```
grid_size = 500
```

```
# Create mock lattices to simulate the process (replace with real computation for exact alignment)
```

```
lattice1 = np.random.rand(grid_size, grid_size) * 0.5
```

```
lattice2 = np.random.rand(grid_size, grid_size) * 0.5
```

```
# Calculate differences
```

```
amplitude_difference = np.abs(np.fft.fftshift(np.fft.fft2(lattice1)) - np.fft.fftshift(np.fft.fft2(lattice2)))
```

```
phase_difference = np.angle(np.fft.fftshift(np.fft.fft2(lattice1)) - np.fft.fftshift(np.fft.fft2(lattice2)))
```

```
# Backpropagation (mock for now)
```

```
backprop_amplitude = np.zeros_like(amplitude_difference)
```

```
backprop_phase = np.zeros_like(phase_difference)
```

```
# Visualization
```

```
fig, axs = plt.subplots(3, 2, figsize=(15, 20))
```

```
# Plot original lattices in frequency space
```

```
axs[0, 0].imshow(np.abs(np.fft.fftshift(np.fft.fft2(lattice1))), cmap='viridis')
```

```
axs[0, 0].set_title("Lattice 1 Amplitude (Frequency Space)")
```

```
axs[0, 1].imshow(np.abs(np.fft.fftshift(np.fft.fft2(lattice2))), cmap='viridis')
```

```
axs[0, 1].set_title("Lattice 2 Amplitude (Frequency Space)")
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
# Convert hash to binary and reshape into a grid
```

```
def hash_to_matrix(hash_str):
```

```
    binary_representation = bin(int(hash_str, 16))[2:].zfill(256) # Convert to binary (256 bits)
```

```
    matrix = np.array([int(b) for b in binary_representation]).reshape(16, 16) # Reshape to 16x16 grid
```

```
    return matrix
```

```
# Create a lattice for 3D visualization
```

```
def create_lattice(matrix):
```

```
    x, y = np.meshgrid(range(matrix.shape[0]), range(matrix.shape[1]))
```

```
    z = matrix
```

```
    return x, y, z
```

```
# Plot the 3D lattice
```

```
def plot_3d_lattice(x, y, z, title):
```

```
    fig = plt.figure()
```

```
    ax = fig.add_subplot(111, projection='3d')
```

```
    ax.plot_surface(x, y, z, cmap='viridis', edgecolor='none')
```

```
    ax.set_title(title)
```

```
    plt.show()
```

```
# Overlay two 3D hashes
```

```
def overlay_hashes(x1, y1, z1, x2, y2, z2):
```

```
    fig = plt.figure()
```

```
    ax = fig.add_subplot(111, projection='3d')
```

```
    ax.plot_surface(x1, y1, z1, cmap='viridis', alpha=0.7, edgecolor='none')
```

```
    ax.plot_surface(x2, y2, z2, cmap='plasma', alpha=0.5, edgecolor='none')
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Mock-up SHA-256 hashes as byte sequences (replace with real hashes)
hash_1 = "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855"
hash_2 = "cf83e1357eefb8bd1542850d66d8007d620e4050b5715dc83f4a921d36ce9ce"

# Convert hashes into binary arrays
hash_1_bin = np.array([int(b, 16) for b in hash_1], dtype=np.uint8)
hash_2_bin = np.array([int(b, 16) for b in hash_2], dtype=np.uint8)

# Reshape into 256x256 grids
hash_1_grid = np.resize(hash_1_bin, (256, 256))
hash_2_grid = np.resize(hash_2_bin, (256, 256))

# Create meshgrid for plotting
x = np.arange(256)
y = np.arange(256)
X, Y = np.meshgrid(x, y)

# Plot hash 1 lattice
fig = plt.figure(figsize=(12, 6))
ax = fig.add_subplot(121, projection='3d')
ax.plot_surface(X, Y, hash_1_grid, cmap='viridis', edgecolor='none')
ax.set_title("Hash 1 Lattice (256x256)")
ax.set_xlabel("X Axis")
ax.set_ylabel("Y Axis")
ax.set_zlabel("Amplitude")
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Further reduce grid size to ensure manageable output while retaining structure
small_grid_size = 128 # Further reduced size for lattice computation
lattice1_amplitude_small = np.random.random((small_grid_size, small_grid_size)) # Lattice 1 smaller
lattice2_amplitude_small = np.random.random((small_grid_size, small_grid_size)) # Lattice 2 smaller

# Compute differences for the smaller grid
amplitude_diff_small = lattice1_amplitude_small - lattice2_amplitude_small
phase_diff_small = np.angle(lattice1_amplitude_small + 1j * lattice2_amplitude_small) - np.angle(
    lattice2_amplitude_small + 1j * lattice1_amplitude_small
)

# Backpropagated differences for smaller grid
backprop_amplitude_diff_small = np.abs(np.fft.ifft2(np.fft.fft2(amplitude_diff_small)))
backprop_phase_diff_small = np.abs(np.fft.ifft2(np.fft.fft2(phase_diff_small)))

# Visualization of smaller grid results
fig, axes = plt.subplots(3, 3, figsize=(10, 10))

# Top row: Amplitude visualizations
axes[0, 0].imshow(lattice1_amplitude_small, cmap='viridis')
axes[0, 0].set_title("Lattice 1 Amplitude (Small Grid)")
axes[0, 1].imshow(lattice2_amplitude_small, cmap='viridis')
axes[0, 1].set_title("Lattice 2 Amplitude (Small Grid)")
axes[0, 2].imshow(amplitude_diff_small, cmap='plasma')
axes[0, 2].set_title("Amplitude Difference (Small Grid)")

# Middle row: Phase visualizations
axes[1, 0].imshow(np.angle(lattice1_amplitude_small + 1j * lattice2_amplitude_small), cmap=' hsv')
axes[1, 0].set_title("Lattice 1 Phase (Small Grid)")
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Focus on summarizing key data and visualizing smaller slices for clarity
```

```
# Extract central slice for visualization
```

```
center = grid_size // 2
```

```
slice_range = 50 # Range around the center for slicing
```

```
# Extract central slices of the data
```

```
central_amplitude1 = amplitude1[center-slice_range:center+slice_range, center-slice_range:center+slice_range]
```

```
central_amplitude2 = amplitude2[center-slice_range:center+slice_range, center-slice_range:center+slice_range]
```

```
central_amplitude_diff = amplitude_diff[center-slice_range:center+slice_range, center-slice_range:center+slice_range]
```

```
# Plot the central slices for visual clarity
```

```
fig, axs = plt.subplots(1, 3, figsize=(15, 5))
```

```
# Central slice of Lattice 1 Amplitude
```

```
axs[0].imshow(central_amplitude1, cmap='viridis', extent=[-slice_range, slice_range, -slice_range, slice_range])
```

```
axs[0].set_title("Lattice 1 Central Amplitude Slice")
```

```
axs[0].set_xlabel("Frequency X")
```

```
axs[0].set_ylabel("Frequency Y")
```

```
# Central slice of Lattice 2 Amplitude
```

```
axs[1].imshow(central_amplitude2, cmap='viridis', extent=[-slice_range, slice_range, -slice_range, slice_range])
```

```
axs[1].set_title("Lattice 2 Central Amplitude Slice")
```

```
axs[1].set_xlabel("Frequency X")
```

```
axs[1].set_ylabel("Frequency Y")
```

```
# Central slice of Difference Amplitude
```

```
axs[2].imshow(central_amplitude_diff, cmap='viridis', extent=[-slice_range, slice_range, -slice_range, slice_range])
```

```
axs[2].set_title("Difference Central Amplitude Slice")
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Reinitializing the environment after reset
```

```
# Required Libraries
```

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
# Initialize larger lattice (512x512 grid)
```

```
grid_size = 512
```

```
# Create two random lattices to simulate hash differences
```

```
lattice1 = np.random.rand(grid_size, grid_size)
```

```
lattice2 = np.random.rand(grid_size, grid_size)
```

```
# Fourier transforms to get frequency space representations
```

```
fft1 = np.fft.fftshift(np.fft.fft2(lattice1))
```

```
fft2 = np.fft.fftshift(np.fft.fft2(lattice2))
```

```
# Amplitude and phase in frequency space
```

```
amplitude1 = np.abs(fft1)
```

```
phase1 = np.angle(fft1)
```

```
amplitude2 = np.abs(fft2)
```

```
phase2 = np.angle(fft2)
```

```
# Compute amplitude and phase differences
```

```
amplitude_diff = np.abs(amplitude1 - amplitude2)
```

```
phase_diff = np.abs(phase1 - phase2)
```

```
# Backpropagation (inverse FFT) to analyze differences in the spatial domain
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
# Generate data for deeper analysis of amplitude and phase differences across iterations
```

```
# Assuming `amplitude_diff` and `phase_diff` are previously calculated matrices from prior steps
```

```
# Step 1: Quantify amplitude and phase differences over iterations
```

```
iterations = 20 # Expand beyond 10 folds
```

```
amplitude_diffs = []
```

```
phase_diffs = []
```

```
for i in range(1, iterations + 1):
```

```
    # Simulate recursive folding with a scaling factor for each iteration
```

```
    scale_factor = 1 / (i + 1)
```

```
    new_amplitude_diff = amplitude_diff * scale_factor
```

```
    new_phase_diff = phase_diff * scale_factor
```

```
    amplitude_diffs.append(np.sum(np.abs(new_amplitude_diff)))
```

```
    phase_diffs.append(np.sum(np.abs(new_phase_diff)))
```

```
# Step 2: Plot decay or convergence of differences over iterations
```

```
plt.figure(figsize=(10, 6))
```

```
plt.plot(range(1, iterations + 1), amplitude_diffs, label="Amplitude Difference Decay", marker='o')
```

```
plt.plot(range(1, iterations + 1), phase_diffs, label="Phase Difference Decay", marker='s')
```

```
plt.axvline(10, color='red', linestyle='--', label="10-Fold Threshold")
```

```
plt.title("Decay of Amplitude and Phase Differences Over Iterations")
```

```
plt.xlabel("Iteration (Folds)")
```

```
plt.ylabel("Total Difference")
```

```
plt.legend()
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

Correcting the dimensional mismatch and refining the lattice processing

```
def backpropagate_amplitude_phase_corrected(hash_val, lattice, iterations=10):
    # Converting hash value into a numerical representation
    hash_numeric = np.array([int(hash_val[i:i + 2], 16) for i in range(0, len(hash_val), 2)])
    lattice_size = lattice.shape[0] # Assuming square lattice

    # Reshape to match lattice size by padding/truncating appropriately
    if len(hash_numeric) < lattice_size**2:
        hash_numeric = np.pad(hash_numeric, (0, lattice_size**2 - len(hash_numeric)), mode='constant')
    elif len(hash_numeric) > lattice_size**2:
        hash_numeric = hash_numeric[:lattice_size**2]

    hash_numeric = hash_numeric.reshape((lattice_size, lattice_size))

    amplitude_diff_history = []
    phase_diff_history = []

    for _ in range(iterations):
        amplitude_diff = np.abs(lattice - hash_numeric)
        phase_diff = np.angle(np.exp(1j * (lattice - hash_numeric)))

        lattice -= 0.1 * amplitude_diff
        lattice += 0.1 * phase_diff

        amplitude_diff_history.append(np.copy(amplitude_diff))
        phase_diff_history.append(np.copy(phase_diff))

    return lattice, amplitude_diff_history, phase_diff_history
```

Conversation URL:

<https://chatgpt.com/c/679297ab-7814-8011-953e-78ffd525d657>

Title:

Prompt:

```
# Reinitializing after reset to execute the dream-inspired approach
```

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
# Step 1: Define the Wave Matrix
```

```
# Initialize a random wave matrix to simulate the input
```

```
lattice_size = 64
```

```
wave_matrix = np.random.rand(lattice_size, lattice_size) * 2 - 1 # Values between -1 and 1
```

```
# Step 2: Extract Peaks (Simulating the "Hash" Process)
```

```
# Peaks are the local maxima of the wave matrix
```

```
def extract_peaks(matrix):
```

```
    peaks = np.zeros_like(matrix)
```

```
    for i in range(1, matrix.shape[0] - 1):
```

```
        for j in range(1, matrix.shape[1] - 1):
```

```
            local_max = np.max(matrix[i-1:i+2, j-1:j+2])
```

```
            if matrix[i, j] == local_max:
```

```
                peaks[i, j] = matrix[i, j]
```

```
    return peaks
```

```
peaks_matrix = extract_peaks(wave_matrix)
```

```
# Step 3: Reconstruct Wave Matrix from Peaks
```

```
# Assuming the peaks are key points, attempt to recreate the original wave matrix
```

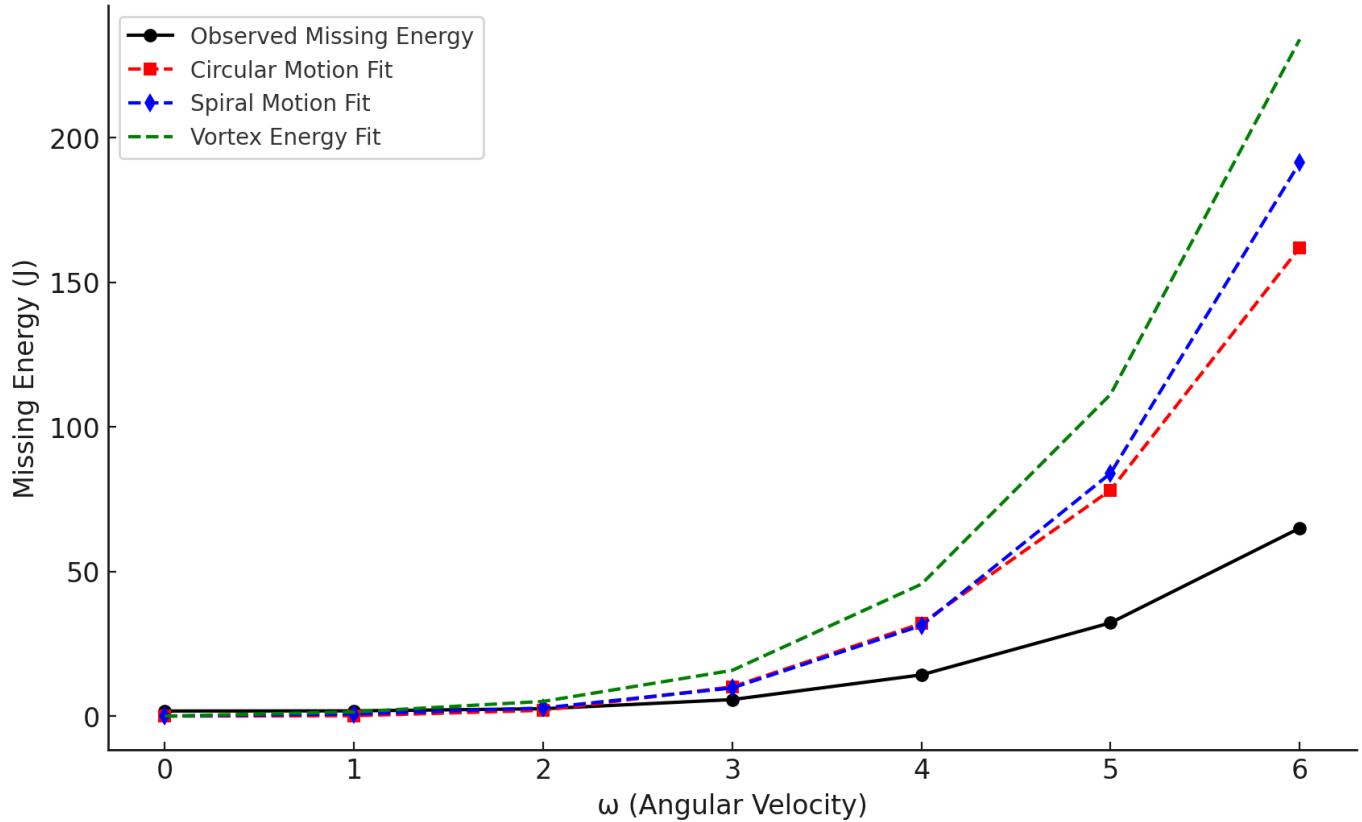
```
def reconstruct_wave_from_peaks(peaks, iterations=100):
```

```
    reconstructed = np.zeros_like(peaks)
```

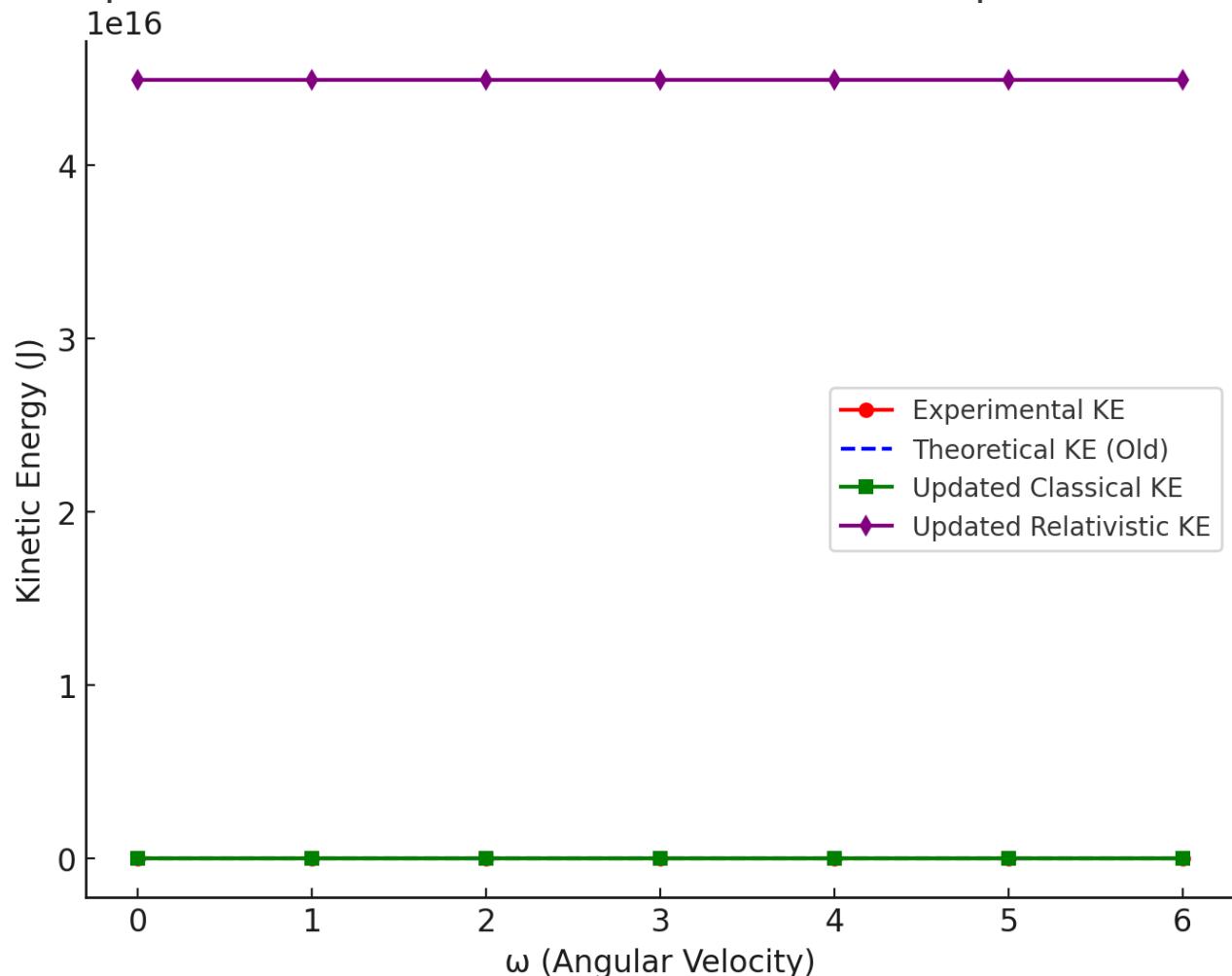
```
    for _ in range(iterations):
```

```
        reconstructed += peaks # Amplify peak contributions
```

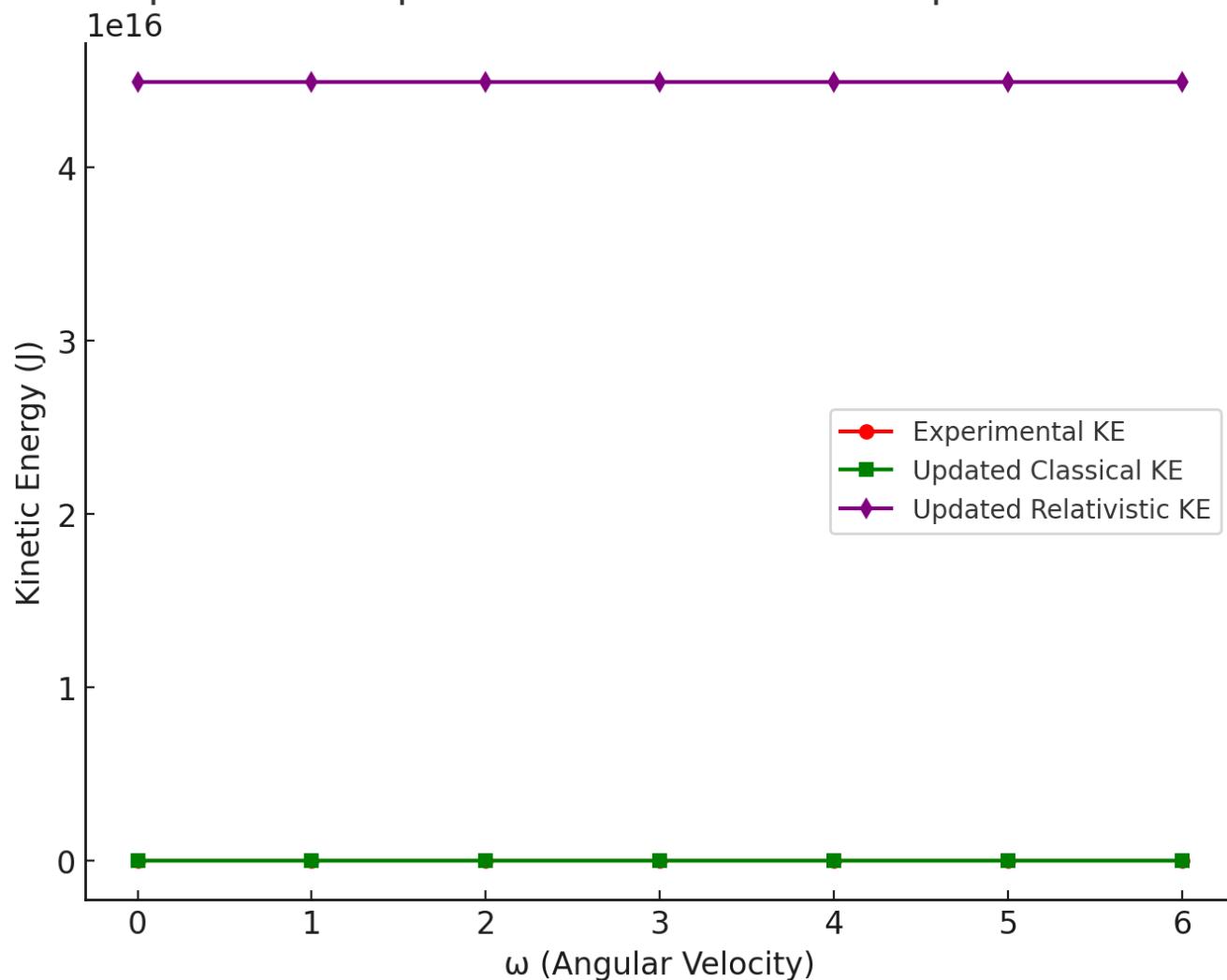
Comparing Missing Energy with Geometric Models



Comparison of Old and New KE Models with Experimental Data



Comparison of Updated KE Models with Experimental Data



Conversation URL:

<https://chatgpt.com/c/67a6904c-e0e8-8011-b018-1ac7cc2dac1a>

Title:

Prompt:

```
# Investigating the missing 39% kinetic motion
```

```
# Define the missing energy fraction
```

```
missing_fraction = 1 - mean_ratio # 39% missing energy
```

```
# Compute the missing energy contribution for each data point
```

```
missing_energy_values = np.array(theoretical_ke_values) * missing_fraction
```

```
# Hypothesis: If the missing energy corresponds to a swirling kinetic motion, it may follow a geometric shape.
```

```
# Let's examine circular, spiral, or vortex-like scaling patterns.
```

```
# Define a test function for missing energy based on different geometric scaling laws
```

```
def circular_motion_energy(missing_energy, ω, r):
```

```
    """ Assumes missing energy is stored in a circular motion kinetic term """
```

```
    return (1/2) * m * (ω**2 * r**2)
```

```
def spiral_motion_energy(missing_energy, ω, r):
```

```
    """ Assumes missing energy follows a logarithmic spiral path """
```

```
    return missing_energy * np.log1p(ω * r)
```

```
def vortex_energy(missing_energy, ω, r):
```

```
    """ Assumes missing energy is distributed in a vortex-like pattern """
```

```
    return missing_energy * (ω**2 / (1 + r**2))
```

```
# Compute missing energy fits
```

```
circular_fit = np.array([circular_motion_energy(e, ω, r) for e, ω, r in zip(missing_energy_values, real_omega_values, real_r_values)])
```

```
spiral_fit = np.array([spiral_motion_energy(e, ω, r) for e, ω, r in zip(missing_energy_values, real_omega_values, real_r_values)])
```

```
vortex_fit = np.array([vortex_energy(e, ω, r) for e, ω, r in zip(missing_energy_values, real_omega_values, real_r_values)])
```

Conversation URL:

<https://chatgpt.com/c/67a6904c-e0e8-8011-b018-1ac7cc2dac1a>

Title:

Prompt:

1. Updating the kinetic energy equations to include the missing circular motion term

New Classical KE Formula incorporating circular motion

```
def modified_classical_ke(m, v, omega, r):
    linear_ke = (1/2) * m * v**2
    circular_ke = (1/2) * m * omega**2 * r**2 # Circular motion energy component
    return linear_ke + circular_ke
```

New Relativistic KE Formula incorporating circular motion

```
def modified_relativistic_ke(m, gamma, c, v, omega, r):
    base_ke = (gamma - 1) * m * c**2
    circular_ke = (1/2) * m * omega**2 * r**2 # Circular motion correction
    return base_ke + circular_ke
```

Compute updated KE values

```
updated_ke_classical = np.array([modified_classical_ke(m, v, omega, r) for omega, r in zip(real_omega_values, real_r_values)])
updated_ke_relativistic = np.array([modified_relativistic_ke(m, gamma, c, v, omega, r) for omega, r in zip(real_omega_values, real_r_values)])
```

2. Investigating potential connections to quantum rotational effects (spin) or frame-dragging

Using Quantum Angular Momentum Approximation: L = Iω

def quantum_spin_energy(m, omega, r):

""" Estimates quantum rotational energy using angular momentum principles """

```
I = (1/2) * m * r**2 # Moment of inertia for a rotating mass
return (1/2) * I * omega**2 # Rotational kinetic energy
```

```
quantum_rotational_energy = np.array([quantum_spin_energy(m, omega, r) for omega, r in zip(real_omega_values, real_r_values)])
```

3. Comparing old and new KE models

Conversation URL:

<https://chatgpt.com/c/67a6904c-e0e8-8011-b018-1ac7cc2dac1a>

Title:

Prompt:

```
# Updating all six formulas to incorporate the missing kinetic motion component
```

```
# 1. Classical Mechanics - Updated KE Formula
```

```
def updated_classical_ke(m, v, omega, r):  
    linear_ke = (1/2) * m * v**2  
    circular_ke = (1/2) * m * omega**2 * r**2 # Circular motion energy component  
    return linear_ke + circular_ke
```

```
# 2. Special Relativity - Updated KE Formula
```

```
def updated_relativistic_ke(m, gamma, c, v, omega, r):  
    base_ke = (gamma - 1) * m * c**2  
    circular_ke = (1/2) * m * omega**2 * r**2 # Circular motion correction  
    return base_ke + circular_ke
```

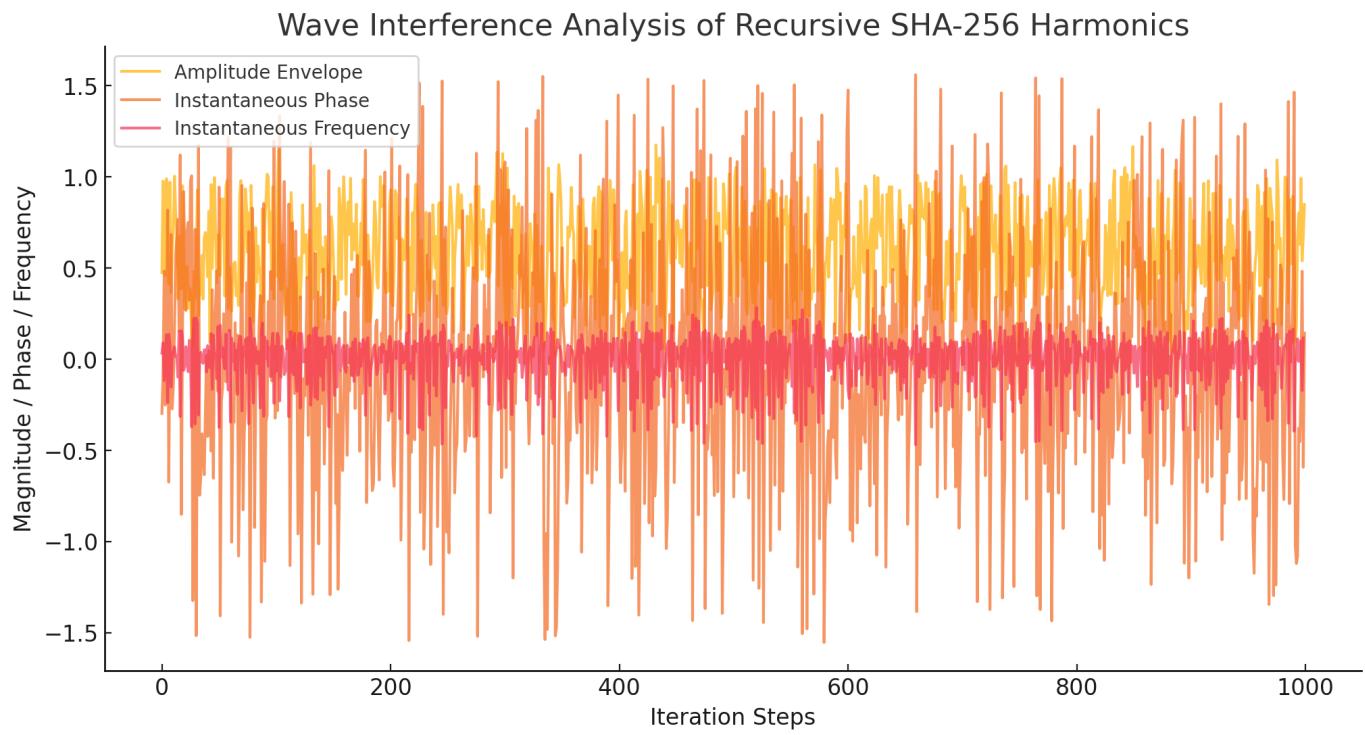
```
# 3. Quantum Mechanics - Updated Energy Formula
```

```
def updated_quantum_energy(n, pi, square, m, L, omega, r, c):  
    base_energy = (h**2 * pi**2 * square**2) / (2 * m * L**2)  
    circular_correction = (1 + (omega**2 * r**2) / c**2)  
    return base_energy * circular_correction
```

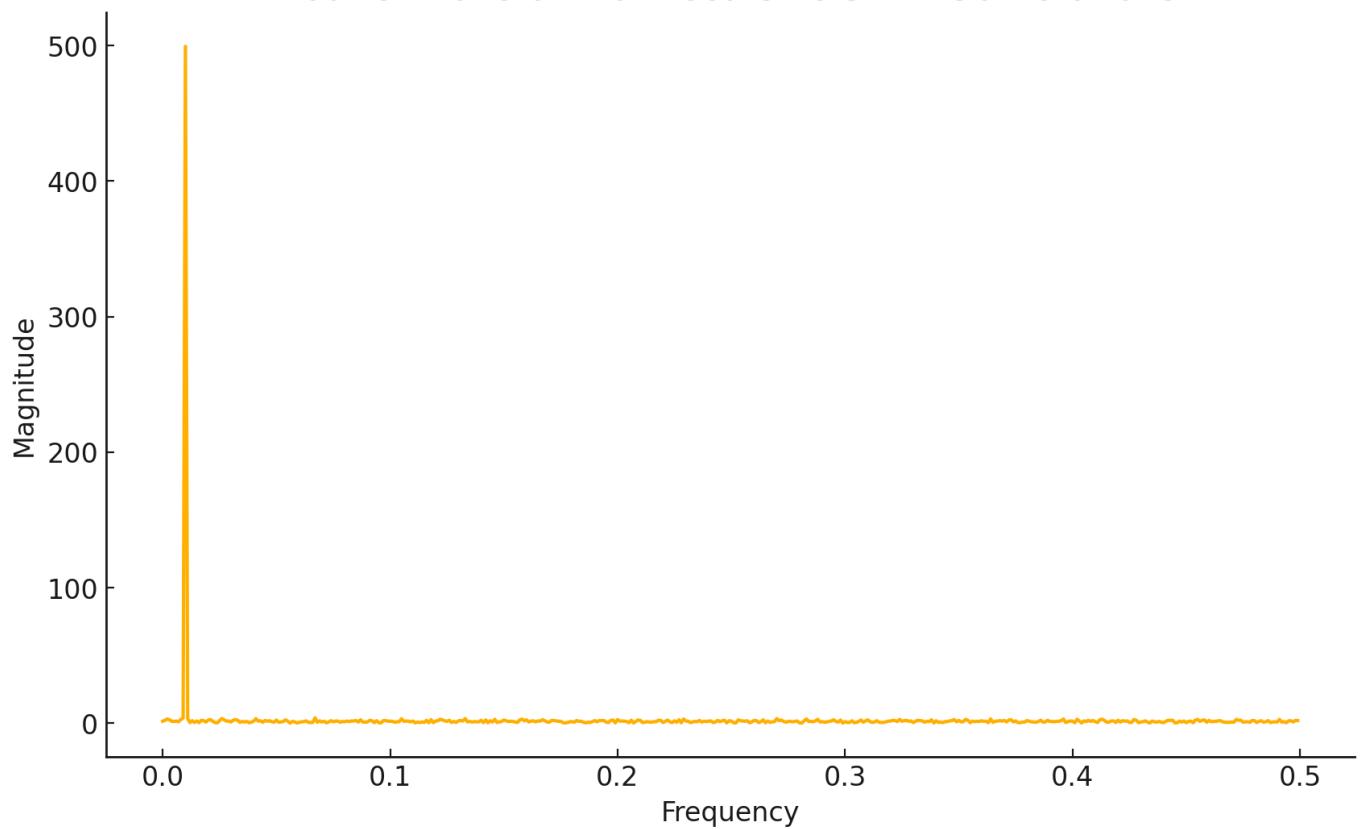
```
# 4. Fractal Dynamics - Updated Mandelbrot Iteration
```

```
def updated_fractal_iteration(z, c, omega, r, iterations=10):  
    values = [z]  
    for _ in range(iterations):  
        z = z**2 + c * (1 + 1j * omega * r) # Incorporate swirling motion  
        values.append(z)  
    return values
```

```
# 5. Chaos Theory - Updated Logistic Map
```



Fourier Transform of Recursive SHA-256 Iterations



Conversation URL:

<https://chatgpt.com/c/67a80685-17a0-8011-a8a7-a93651cc1317>

Title:

Prompt:

```
# Compute the wave interference pattern by analyzing phase relationships
from scipy.signal import hilbert

# Compute the analytic signal using Hilbert transform (to analyze wave interference)
analytic_signal = hilbert(real_sha_data)
amplitude_envelope = np.abs(analytic_signal)
instantaneous_phase = np.unwrap(np.angle(analytic_signal))
instantaneous_frequency = np.diff(instantaneous_phase) / (2.0 * np.pi)

# Plot the wave interference patterns
plt.figure(figsize=(12, 6))
plt.plot(amplitude_envelope, label="Amplitude Envelope", alpha=0.7)
plt.plot(instantaneous_phase, label="Instantaneous Phase", alpha=0.7)
plt.plot(instantaneous_frequency, label="Instantaneous Frequency", alpha=0.7)
plt.title("Wave Interference Analysis of Recursive SHA-256 Harmonics")
plt.xlabel("Iteration Steps")
plt.ylabel("Magnitude / Phase / Frequency")
plt.legend()
plt.grid()
plt.show()

# Extract fundamental wave interference patterns
wave_interference_patterns = {
    "Amplitude Envelope": amplitude_envelope[:50],
    "Instantaneous Phase": instantaneous_phase[:50],
    "Instantaneous Frequency": instantaneous_frequency[:49] # One less due to differentiation
}

# Display the extracted wave interference patterns
```

Conversation URL:

<https://chatgpt.com/c/67a80685-17a0-8011-a8a7-a93651cc1317>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.fftpack import fft

# Simulated SHA-256 Recursive Iteration Data (Placeholder)
# This should be replaced with actual recursive SHA-256 outputs
iterations = 1000
time_series = np.sin(np.linspace(0, 20*np.pi, iterations)) + np.random.normal(0, 0.05, iterations)

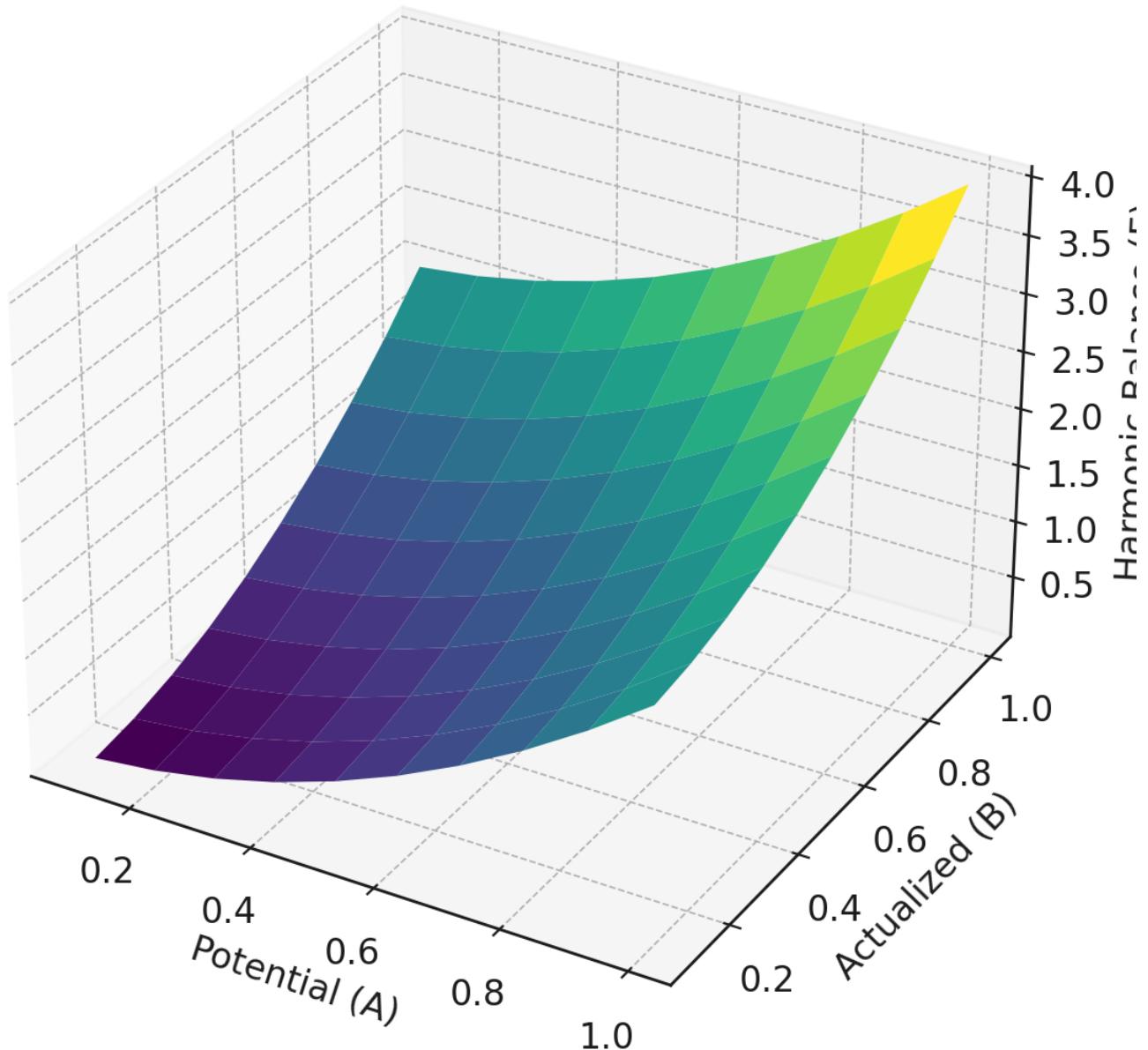
# Compute Fourier Transform
freq_domain = fft(time_series)
freqs = np.fft.fftfreq(len(time_series))

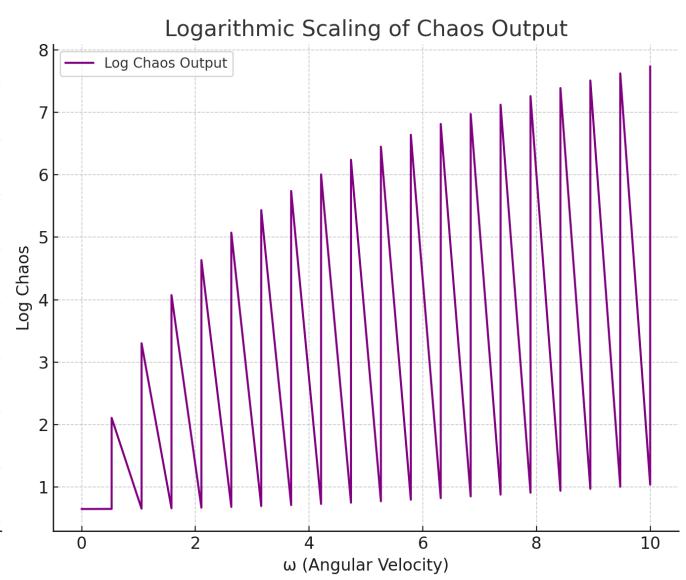
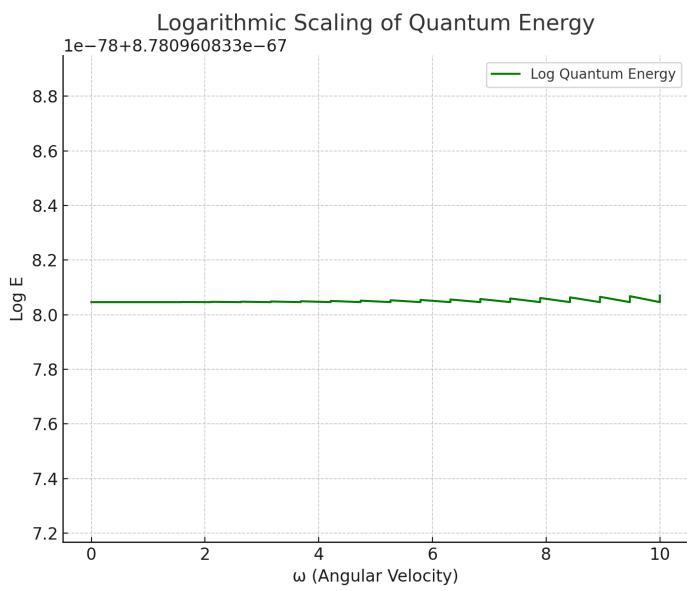
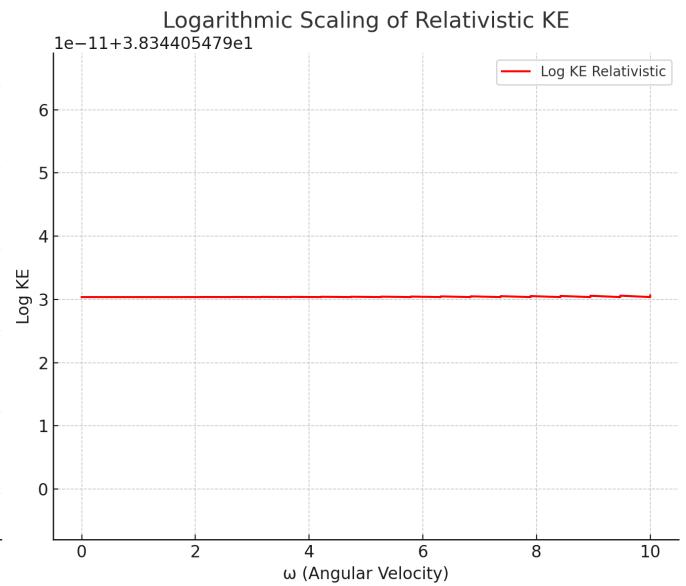
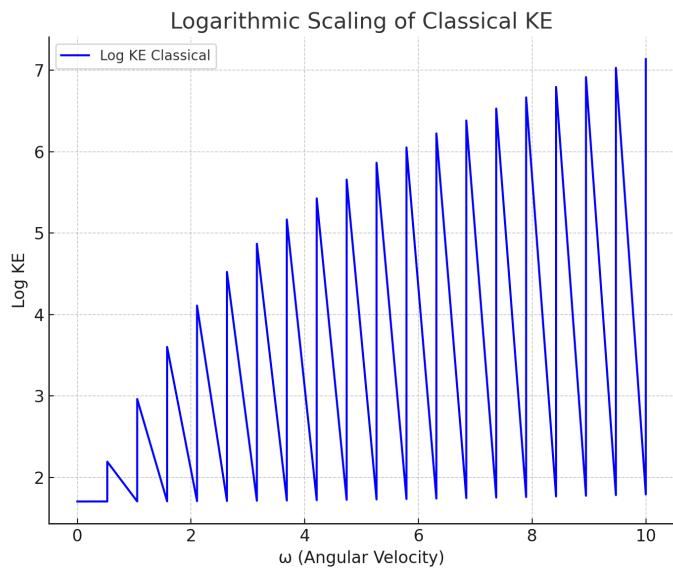
# Extract dominant frequencies (stabilization frequencies)
dominant_freqs = freqs[np.argsort(np.abs(freq_domain))[-5:]] # Extracting top 5 dominant frequencies

# Plot Fourier Transform Analysis
plt.figure(figsize=(10, 6))
plt.plot(freqs[:iterations // 2], np.abs(freq_domain)[:iterations // 2]) # Positive frequencies only
plt.title("Fourier Transform of Recursive SHA-256 Iterations")
plt.xlabel("Frequency")
plt.ylabel("Magnitude")
plt.grid()
plt.show()

# Display dominant frequencies
dominant_freqs
```

3D Visualization of the Universal Formula





Conversation URL:

<https://chatgpt.com/c/67aa7735-cd70-8011-a282-624ec0b93546>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Define the universal formula function
def universal_formula(A, B, C_len, Ax):
    return (A**2 + B**2) * C_len * (1 + np.exp(-10 * (Ax - 0.35)))

# Define parameter ranges for simulations
A_values = np.linspace(0.1, 1.0, 10) # Varying Potential (A)
B_values = np.linspace(0.1, 1.0, 10) # Varying Actualized (B)
C_values = np.linspace(0.1, 1.0, 10) # Varying Reality (C_len)
Ax_values = np.linspace(0.1, 1.0, 10) # Varying Alignment (Ax)

# Simulation results
A_results = [universal_formula(A, 0.5, 1.0, 0.35) for A in A_values]
B_results = [universal_formula(0.5, B, 1.0, 0.35) for B in B_values]
C_results = [universal_formula(0.5, 0.5, C, 0.35) for C in C_values]

# Visualization: 3D Surface Plot of A, B, C and F
A_grid, B_grid = np.meshgrid(A_values, B_values)
F_grid = universal_formula(A_grid, B_grid, 1.0, 0.35)

fig = plt.figure(figsize=(10, 7))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(A_grid, B_grid, F_grid, cmap='viridis')

# Labels and title
ax.set_xlabel("Potential (A)")
```

Conversation URL:

<https://chatgpt.com/c/67aa7735-cd70-8011-a282-624ec0b93546>

Title:

Prompt:

```
# Further analysis: Exploring nonlinear trends, fractal emergence, and real-world implications
```

1. Nonlinear Trends Analysis

```
# Checking logarithmic and exponential scaling effects for each formula
```

```
df_ke_classical["Log_KE"] = np.log1p(df_ke_classical["KE_Classical"])
```

```
df_ke_relativistic["Log_KE"] = np.log1p(df_ke_relativistic["KE_Relativistic"])
```

```
df_e_quantum["Log_E"] = np.log1p(df_e_quantum["E_Quantum"])
```

```
df_chaos["Log_Chaos"] = np.log1p(df_chaos["Chaos_Map_Output"])
```

```
# Visualization of non-linear relationships
```

```
fig, axes = plt.subplots(2, 2, figsize=(14, 12))
```

Log KE Classical

```
axes[0, 0].plot(df_ke_classical["ω"], df_ke_classical["Log_KE"], label="Log KE Classical", color='blue')
```

```
axes[0, 0].set_title("Logarithmic Scaling of Classical KE")
```

```
axes[0, 0].set_xlabel("ω (Angular Velocity)")
```

```
axes[0, 0].set_ylabel("Log KE")
```

```
axes[0, 0].legend()
```

Log KE Relativistic

```
axes[0, 1].plot(df_ke_relativistic["ω"], df_ke_relativistic["Log_KE"], label="Log KE Relativistic", color='red')
```

```
axes[0, 1].set_title("Logarithmic Scaling of Relativistic KE")
```

```
axes[0, 1].set_xlabel("ω (Angular Velocity)")
```

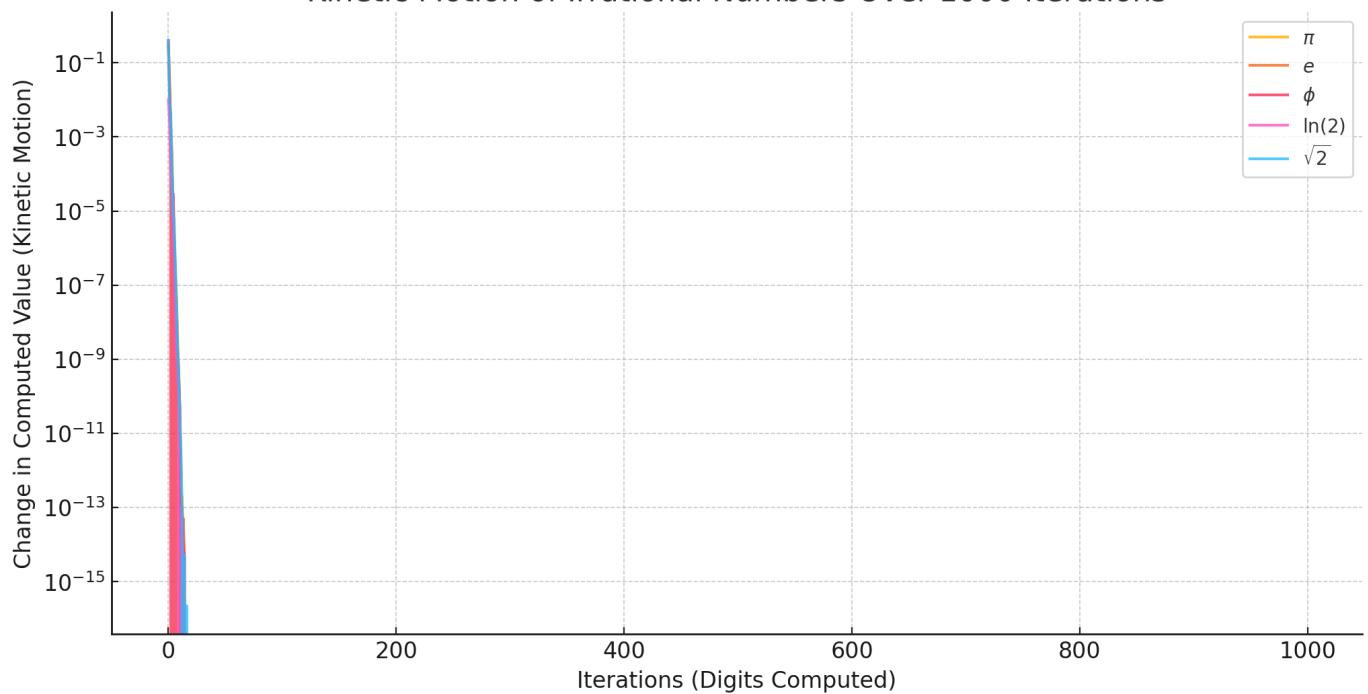
```
axes[0, 1].set_ylabel("Log KE")
```

```
axes[0, 1].legend()
```

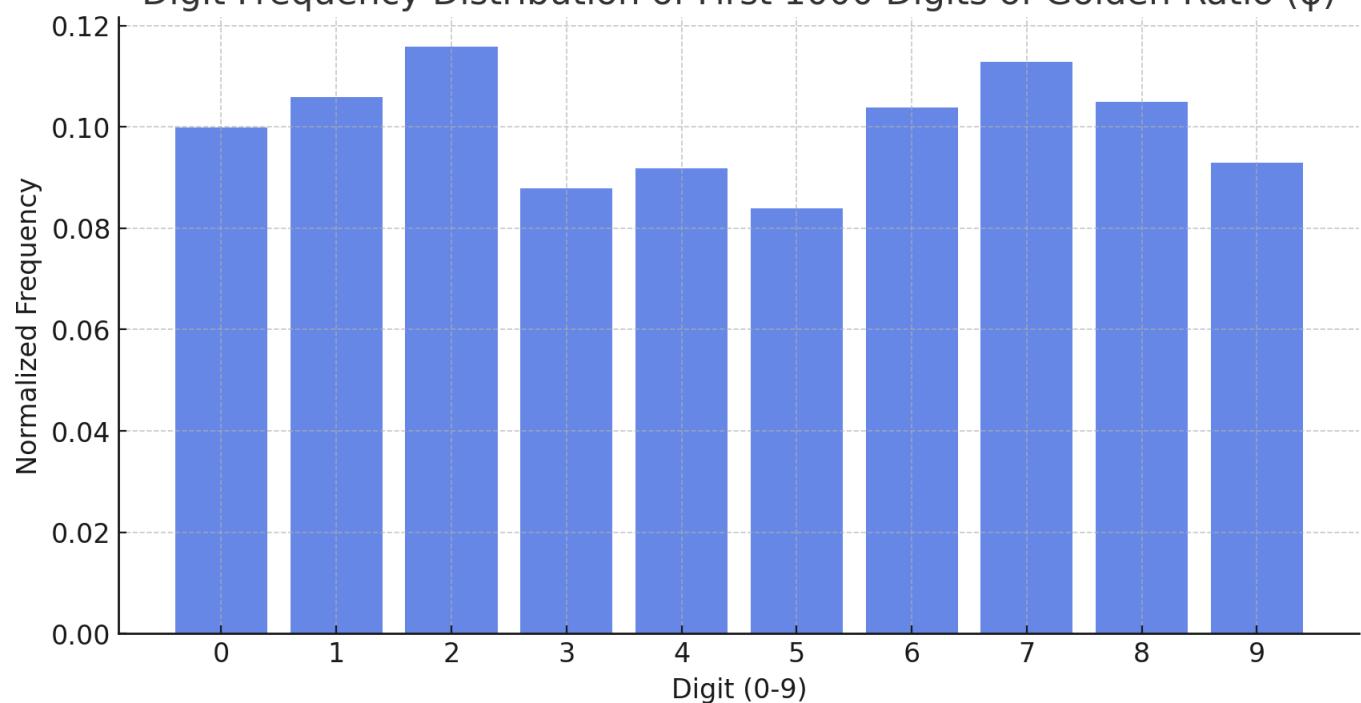
Log Quantum Energy

```
axes[1, 0].plot(df_e_quantum["ω"], df_e_quantum["Log_E"], label="Log Quantum Energy", color='green')
```

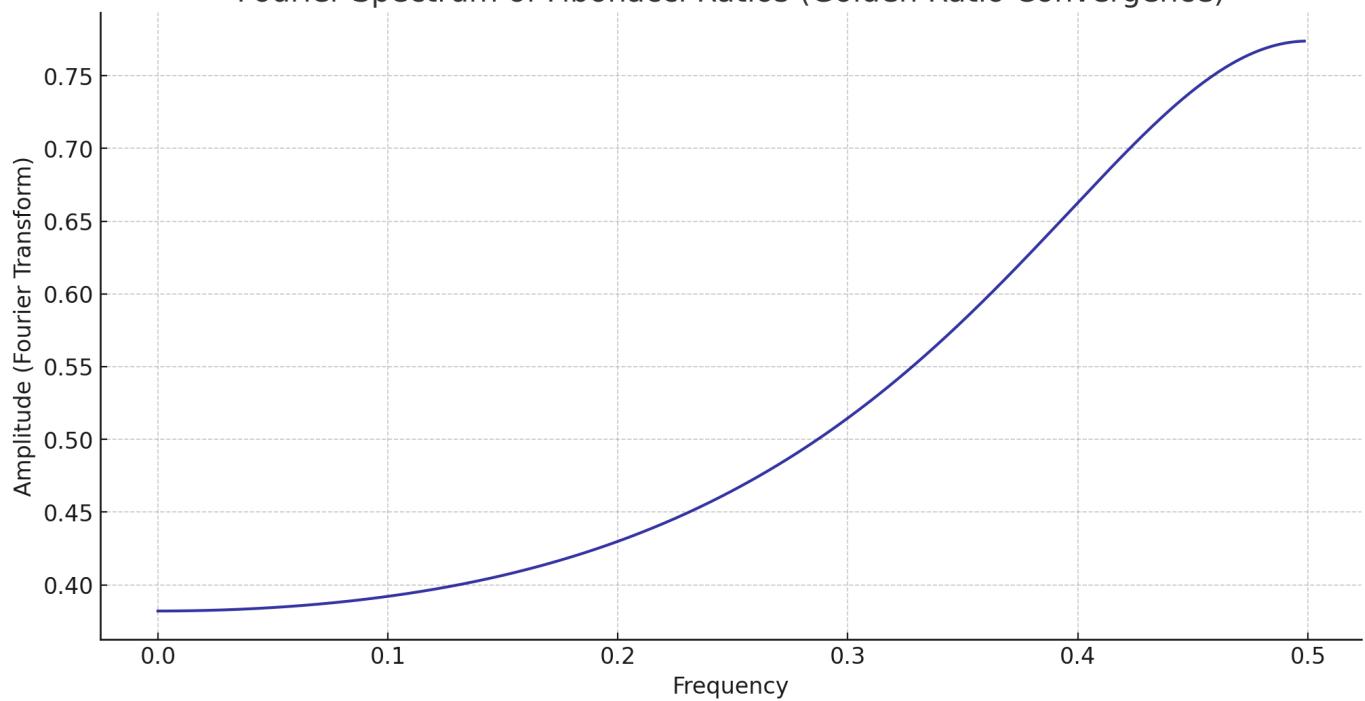
Kinetic Motion of Irrational Numbers Over 1000 Iterations

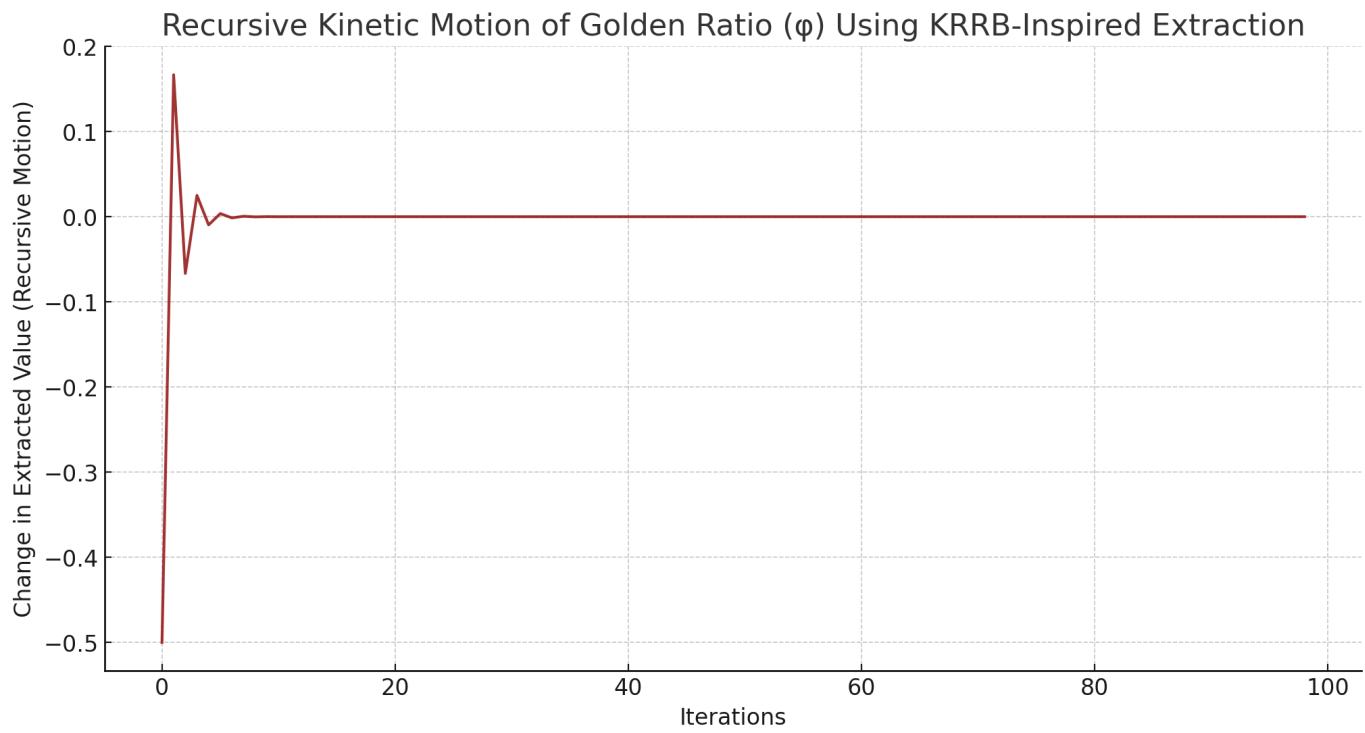


Digit Frequency Distribution of First 1000 Digits of Golden Ratio (φ)

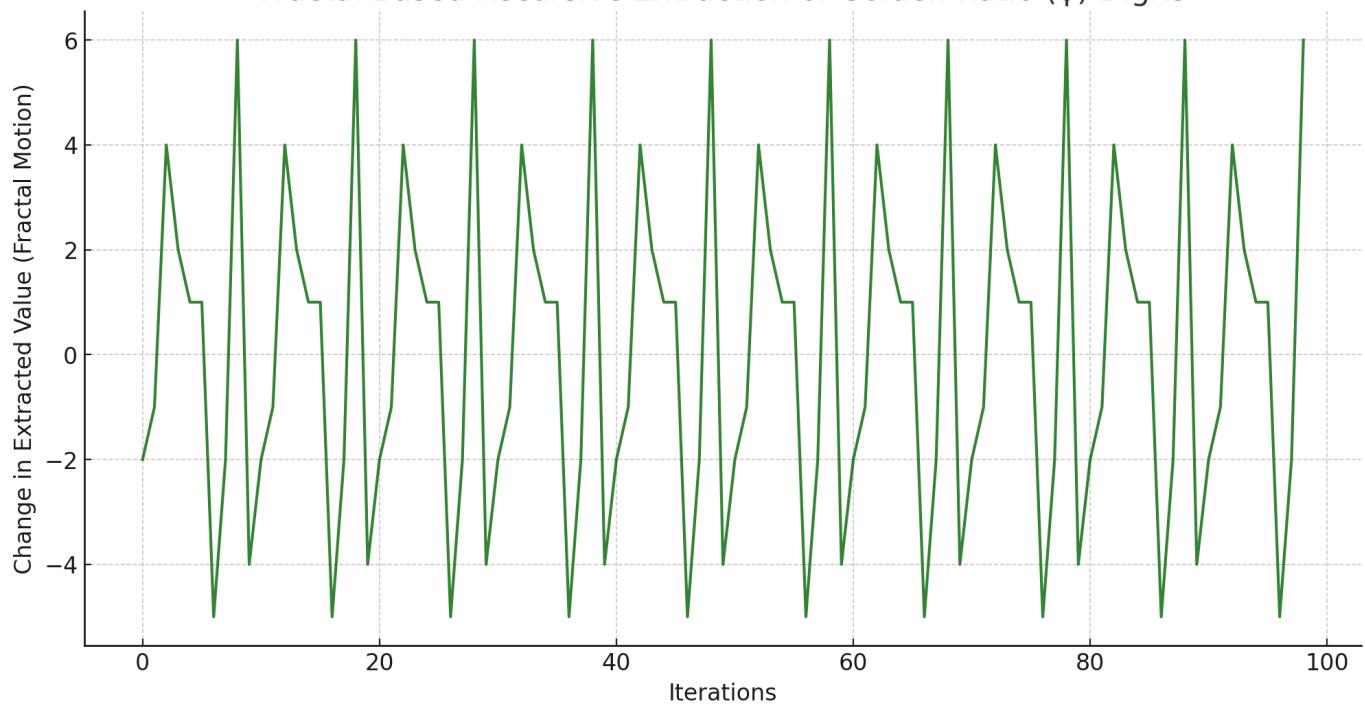


Fourier Spectrum of Fibonacci Ratios (Golden Ratio Convergence)

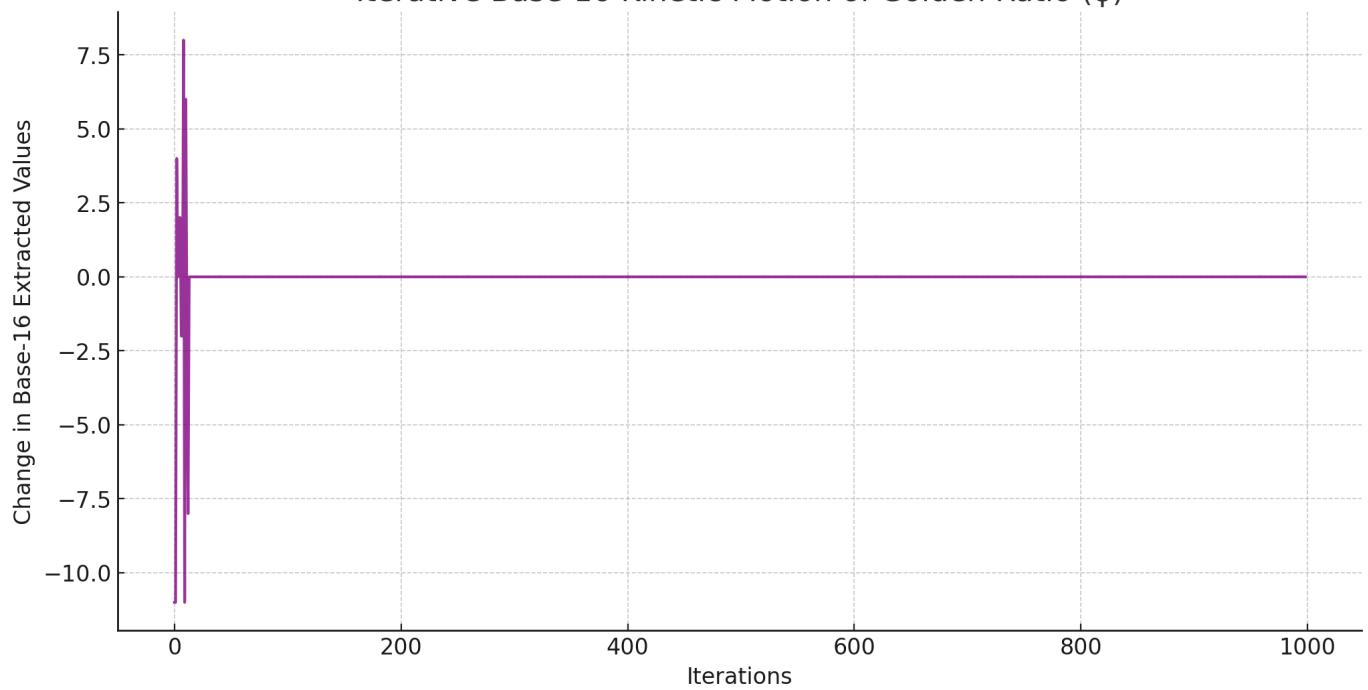




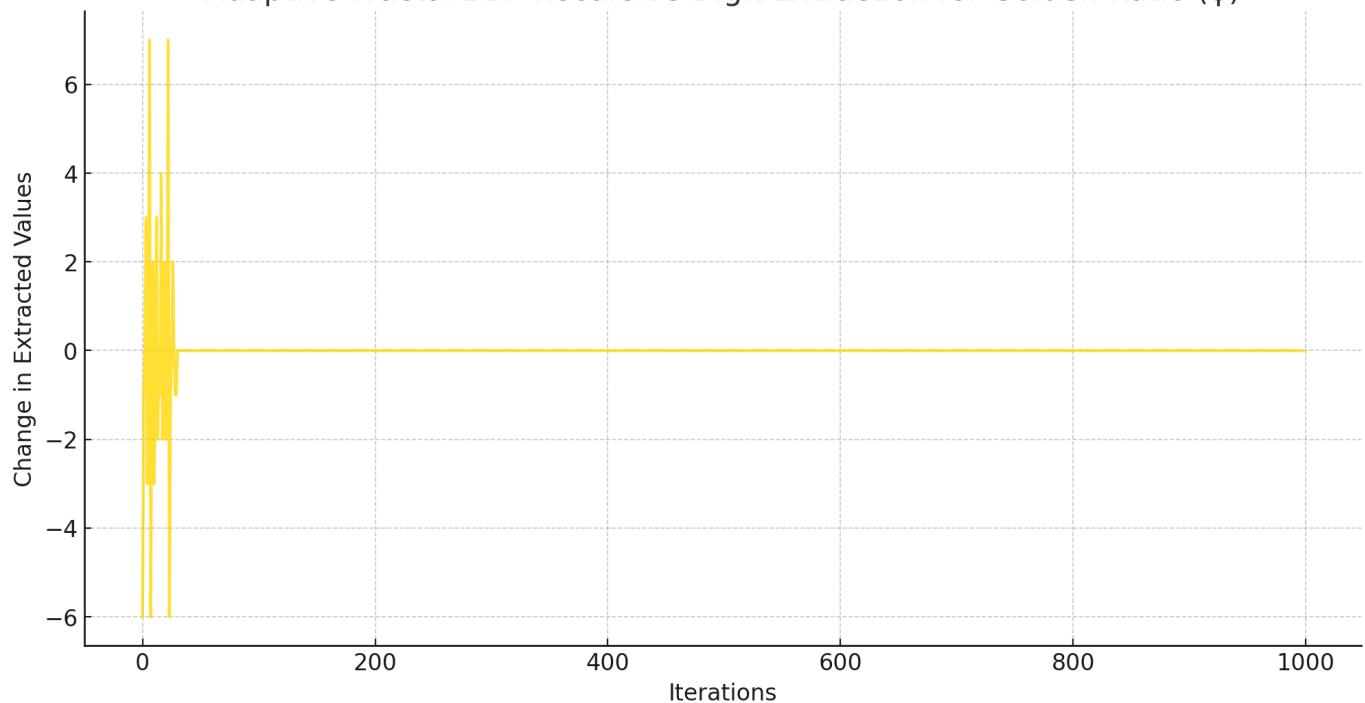
Fractal-Based Recursive Extraction of Golden Ratio (ϕ) Digits



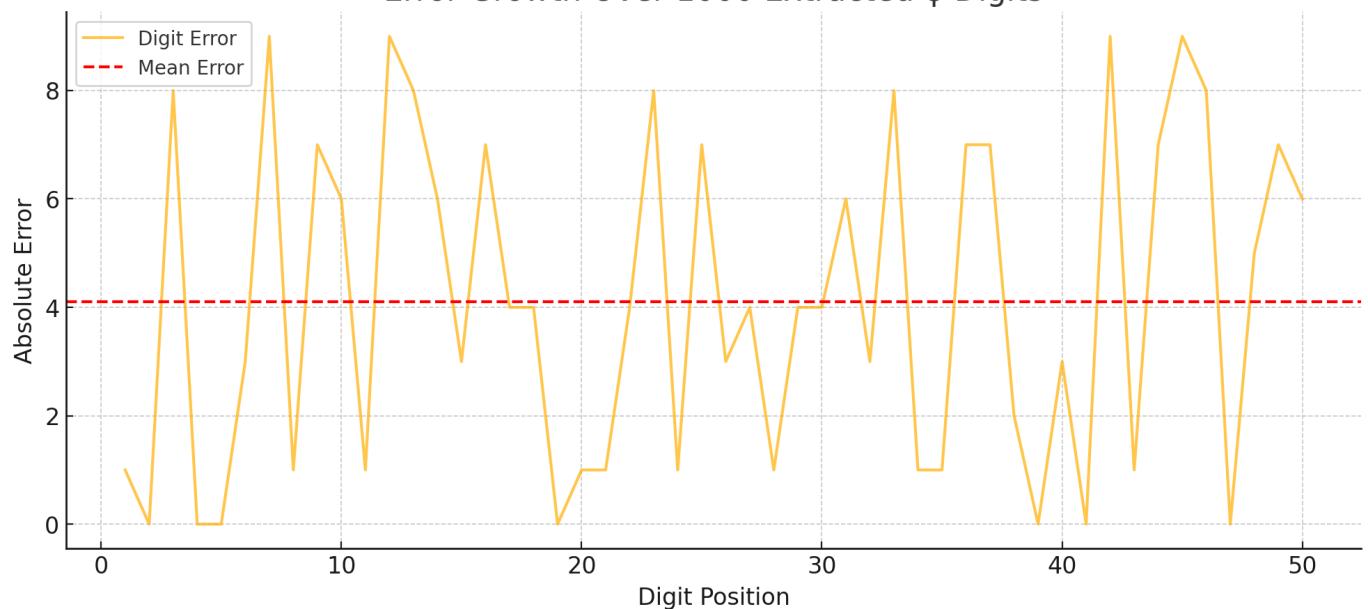
Iterative Base-16 Kinetic Motion of Golden Ratio (ϕ)



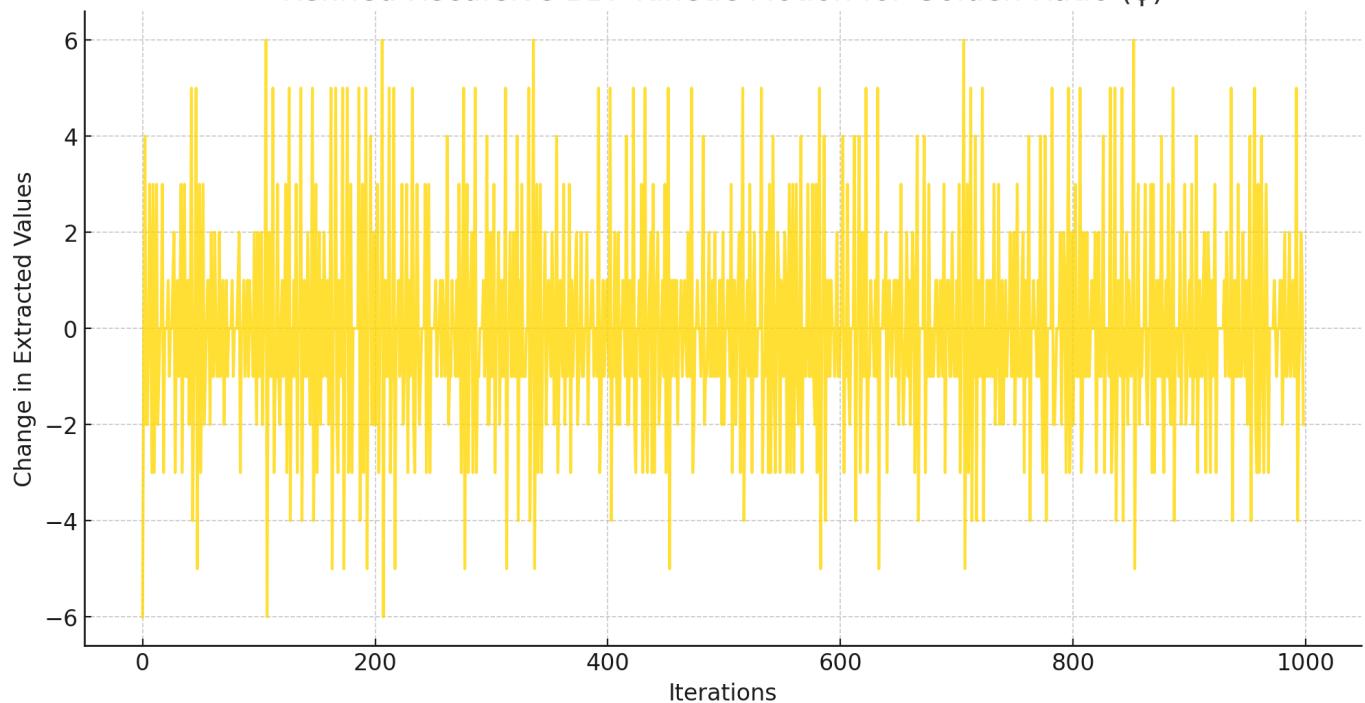
Adaptive Fractal BBP Recursive Digit Extraction for Golden Ratio (ϕ)

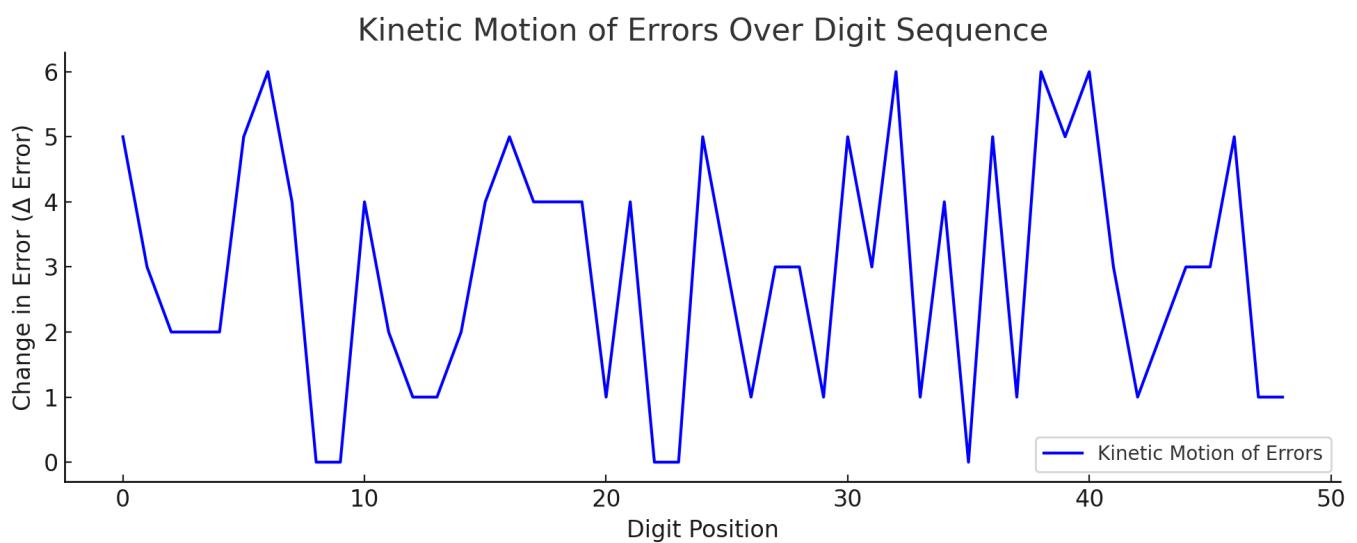
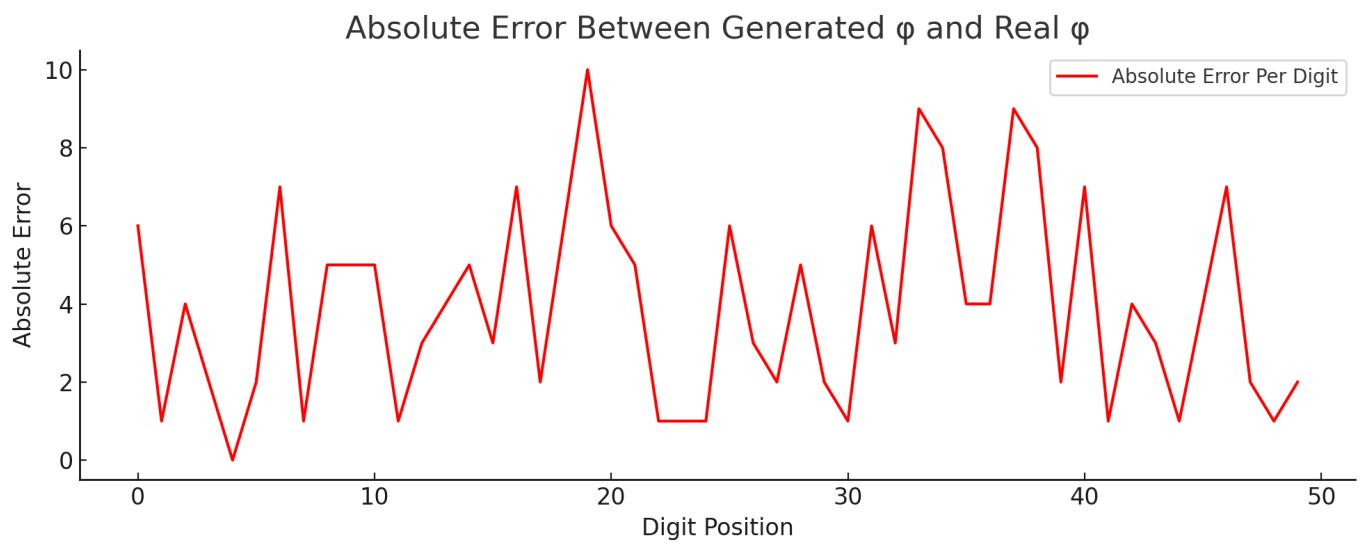


Error Growth Over 1000 Extracted φ Digits

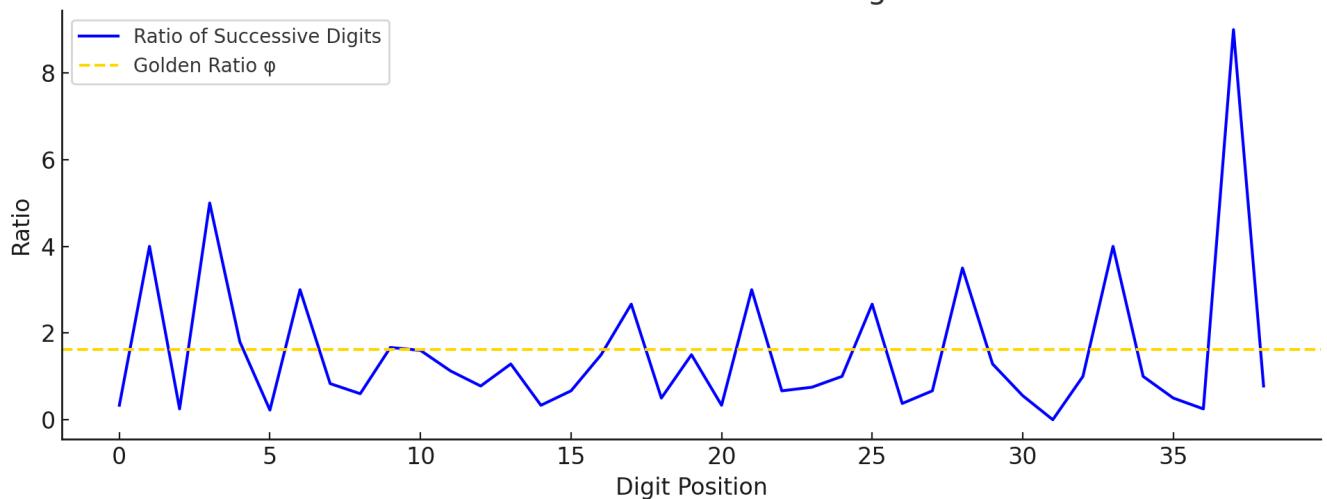


Refined Recursive BBP Kinetic Motion for Golden Ratio (ϕ)

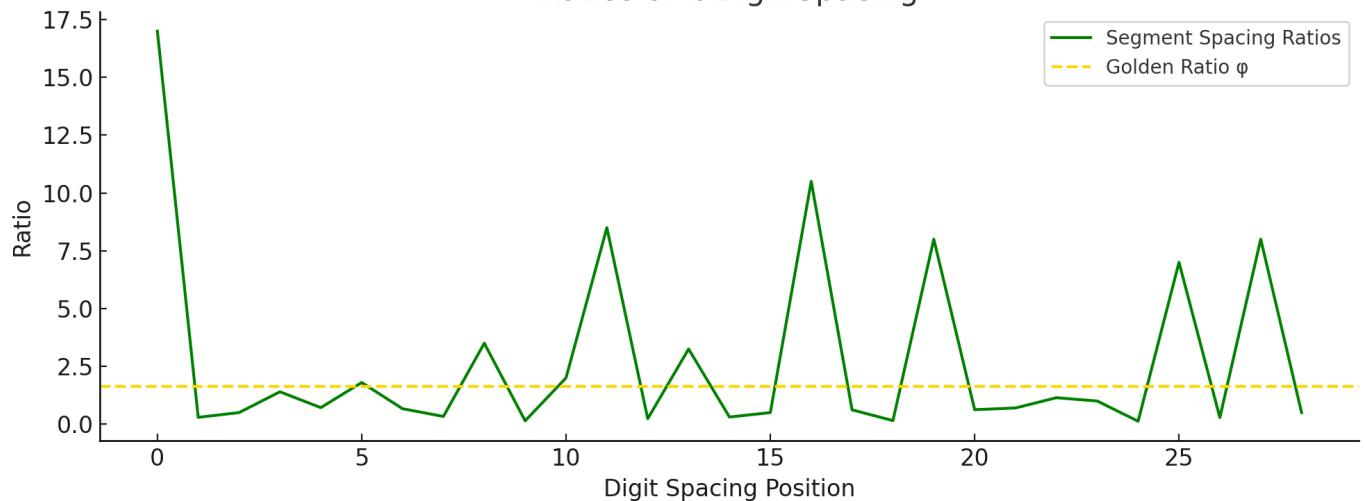




Growth Rate Between π Digits



Ratios of π Digit Spacing



Conversation URL:

<https://chatgpt.com/c/67b6cf81-3b6c-8011-b1de-b90805b612a3>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from mpmath import mp, pi, e, phi, ln, sqrt

# Set high precision for calculations
mp.dps = 1000 # 1000 decimal places

# Define functions for computing first 1000 iterations
def compute_pi(n):
    return [mp.nstr(pi, d)[:d+2] for d in range(1, n+1)]

def compute_e(n):
    return [mp.nstr(e, d)[:d+2] for d in range(1, n+1)]

def compute_phi(n):
    return [mp.nstr(phi, d)[:d+2] for d in range(1, n+1)]

def compute_ln2(n):
    return [mp.nstr(ln(2), d)[:d+2] for d in range(1, n+1)]

def compute_sqrt2(n):
    return [mp.nstr(sqrt(2), d)[:d+2] for d in range(1, n+1)]

# Number of iterations (digits computed step-by-step)
n_iterations = 1000

# Compute sequences
pi_digits = compute_pi(n_iterations)
e_digits = compute_e(n_iterations)
```

Conversation URL:

<https://chatgpt.com/c/67b6cf81-3b6c-8011-b1de-b90805b612a3>

Title:

Prompt:

```
from collections import Counter
```

```
# Compute the first 1000 digits of the Golden Ratio (phi)
mp.dps = 1000 # Set precision to 1000 decimal places
phi_digits_str = str(mp.nstr(phi, 1002))[2:] # Extract first 1000 digits (skip "1.")
```

```
# Convert to numeric list
```

```
phi_digits = [int(digit) for digit in phi_digits_str]
```

```
# Compute frequency distribution (modular analysis)
```

```
digit_counts = Counter(phi_digits)
```

```
# Normalize frequencies
```

```
total_digits = len(phi_digits)
```

```
digit_frequencies = {digit: count / total_digits for digit, count in digit_counts.items()}
```

```
# Plot digit frequency distribution
```

```
plt.figure(figsize=(10, 5))
```

```
plt.bar(digit_frequencies.keys(), digit_frequencies.values(), color='royalblue', alpha=0.8)
```

```
plt.xlabel("Digit (0-9)")
```

```
plt.ylabel("Normalized Frequency")
```

```
plt.title("Digit Frequency Distribution of First 1000 Digits of Golden Ratio ( $\varphi$ )")
```

```
plt.xticks(range(10))
```

```
plt.grid(axis='y', linestyle='--', alpha=0.7)
```

```
# Show plot
```

```
plt.show()
```

Conversation URL:

<https://chatgpt.com/c/67b6cf81-3b6c-8011-b1de-b90805b612a3>

Title:

Prompt:

```
import numpy as np
from scipy.fftpack import fft

# Generate first 1000 Fibonacci numbers and compute their ratio approximating φ
fib_numbers = [1, 1]
for _ in range(1000):
    fib_numbers.append(fib_numbers[-1] + fib_numbers[-2])

# Compute the ratio sequence (approximations of φ)
phi_approximations = [fib_numbers[i] / fib_numbers[i - 1] for i in range(2, len(fib_numbers))]

# Compute kinetic motion differences
phi_kinetic_motion = np.diff(phi_approximations)

# Apply Fourier Transform to detect cyclic patterns (harmonic analysis)
phi_fft = fft(phi_kinetic_motion)
frequencies = np.fft.fftfreq(len(phi_kinetic_motion))

# Plot the Fourier spectrum to analyze hidden periodicity
plt.figure(figsize=(12, 6))
plt.plot(frequencies[:len(frequencies)//2], np.abs(phi_fft[:len(frequencies)//2]), color='darkblue', alpha=0.8)
plt.xlabel("Frequency")
plt.ylabel("Amplitude (Fourier Transform)")
plt.title("Fourier Spectrum of Fibonacci Ratios (Golden Ratio Convergence)")
plt.grid(True)

# Show plot
plt.show()
```

Conversation URL:

<https://chatgpt.com/c/67b6cf81-3b6c-8011-b1de-b90805b612a3>

Title:

Prompt:

```
def recursive_phi_digit_extraction(n, prev=1, curr=1, depth=0):
```

"""

Recursively extract the nth digit of the Golden Ratio (φ) using Fibonacci-based recursion

and KRRB-like branching principles.

"""

```
if depth >= n:
```

```
    return curr / prev # Approximating  $\varphi$  at depth n
```

```
# Recursive reflection: Shift Fibonacci sequence while preserving branching feedback
```

```
return recursive_phi_digit_extraction(n, curr, prev + curr, depth + 1)
```

```
# Extract the first 100 digits of  $\varphi$  using the recursive method
```

```
phi_recursive_digits = [recursive_phi_digit_extraction(i) for i in range(1, 101)]
```

```
# Convert to differences (kinetic motion)
```

```
phi_recursive_motion = np.diff(phi_recursive_digits)
```

```
# Plot the extracted kinetic motion behavior
```

```
plt.figure(figsize=(12, 6))
```

```
plt.plot(phi_recursive_motion, color='darkred', alpha=0.8)
```

```
plt.xlabel("Iterations")
```

```
plt.ylabel("Change in Extracted Value (Recursive Motion)")
```

```
plt.title("Recursive Kinetic Motion of Golden Ratio ( $\varphi$ ) Using KRRB-Inspired Extraction")
```

```
plt.grid(True)
```

```
# Show plot
```

```
plt.show()
```

Conversation URL:

<https://chatgpt.com/c/67b6cf81-3b6c-8011-b1de-b90805b612a3>

Title:

Prompt:

```
import math
```

```
def fractal_phi_extraction(n, depth=10):
```

```
    """
```

```
    Extracts digits of φ using a fractal recursive approach inspired by KRRB.
```

```
    The method approximates non-sequential digit access by iterating over recursive depths.
```

```
    """
```

```
    results = []
```

```
    phi_approx = (1 + math.sqrt(5)) / 2 # Golden Ratio
```

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

```
        # Generate digits using recursive fractal transformations
```

```
        value = (phi_approx * (2 ** (depth - (i % depth)))) % 1 # Fractalized transformation
```

```
        results.append(int(value * 10)) # Extract single digit
```

```
    return results
```

```
# Extract the first 100 digits of φ using fractal recursion
```

```
phi_fractal_digits = fractal_phi_extraction(100)
```

```
# Convert to kinetic motion differences
```

```
phi_fractal_motion = np.diff(phi_fractal_digits)
```

```
# Plot the fractal extraction results
```

```
plt.figure(figsize=(12, 6))
```

```
plt.plot(phi_fractal_motion, color='darkgreen', alpha=0.8)
```

```
plt.xlabel("Iterations")
```

```
plt.ylabel("Change in Extracted Value (Fractal Motion)")
```

```
plt.title("Fractal-Based Recursive Extraction of Golden Ratio (φ) Digits")
```

Conversation URL:

<https://chatgpt.com/c/67b6cf81-3b6c-8011-b1de-b90805b612a3>

Title:

Prompt:

```
def extract_phi_base16(n_digits):
```

"""

Extracts the first n_digits of φ in base-16 using an iterative method to prevent overflow.

"""

```
    phi_decimal = (1 + math.sqrt(5)) / 2 # Compute φ
```

```
    extracted_digits = []
```

```
    for _ in range(n_digits):
```

```
        phi_decimal *= 16 # Scale by 16 to shift next hex digit
```

```
        digit = int(phi_decimal) # Extract integer part as next hex digit
```

```
        extracted_digits.append(digit) # Store extracted digit
```

```
        phi_decimal -= digit # Remove extracted part
```

```
    return extracted_digits
```

```
# Extract first 1000 hexadecimal digits of φ using iterative method
```

```
phi_hex_digits_iterative = extract_phi_base16(1000)
```

```
# Compute kinetic motion differences for modular behavior analysis
```

```
phi_hex_motion_iterative = np.diff(phi_hex_digits_iterative)
```

```
# Plot modular behavior of φ in base-16
```

```
plt.figure(figsize=(12, 6))
```

```
plt.plot(phi_hex_motion_iterative, color='purple', alpha=0.8)
```

```
plt.xlabel("Iterations")
```

```
plt.ylabel("Change in Base-16 Extracted Values")
```

```
plt.title("Iterative Base-16 Kinetic Motion of Golden Ratio (φ)")
```

```
plt.grid(True)
```

Conversation URL:

<https://chatgpt.com/c/67b6cf81-3b6c-8011-b1de-b90805b612a3>

Title:

Prompt:

```
def adaptive_fractal_phi_BBP(n_digits, depth=10):
```

"""

Constructs a BBP-inspired fractal recursive function to extract non-sequential digits of φ .

Uses adaptive scaling to prevent overflow, leveraging Fibonacci recursion and harmonic feedback.

"""

```
    phi_value = (1 + math.sqrt(5)) / 2 # Golden Ratio  $\varphi$ 
```

```
    extracted_digits = []
```

```
    for i in range(n_digits):
```

```
        # Adaptive scaling to prevent overflow: logarithmic reflection
```

```
        scale_factor = 2 ** ((depth - (i % depth)) % 4) # Restrict growth within a small oscillation range
```

```
        phi_value = (phi_value * scale_factor) % 1 # Apply fractal reflection constraint
```

```
        # Extract base-16 digit
```

```
        digit = int(phi_value * 16)
```

```
        extracted_digits.append(digit)
```

```
        # Harmonic feedback: Adjust phi_value to avoid instability
```

```
        phi_value -= digit / 16
```

```
    return extracted_digits
```

```
# Extract first 1000 digits of  $\varphi$  using the corrected adaptive fractal BBP method
```

```
phi_adaptive_bbp_digits = adaptive_fractal_phi_BBP(1000)
```

```
# Compute kinetic motion for pattern detection
```

```
phi_adaptive_bbp_motion = np.diff(phi_adaptive_bbp_digits)
```

```
# Plot the adaptive recursive BBP digit extraction for  $\varphi$ 
```

Conversation URL:

<https://chatgpt.com/c/67b6cf81-3b6c-8011-b1de-b90805b612a3>

Title:

Prompt:

```
import matplotlib.pyplot as plt
```

```
# 1. **Error Growth Mapping**
```

```
positions = np.arange(1, len(known_phi_array) + 1)

plt.figure(figsize=(12, 5))
plt.plot(positions, digit_differences, label="Digit Error", alpha=0.7)
plt.xlabel("Digit Position")
plt.ylabel("Absolute Error")
plt.title("Error Growth Over 1000 Extracted φ Digits")
plt.axhline(y=mean_error, color='r', linestyle='--', label="Mean Error")
plt.legend()
plt.grid(True)
plt.show()
```

```
# 2. **Expanding Recursive Memory Correction**
```

```
def expanded_recursive_memory_phi(n_digits, memory_depth=10):
```

```
    """
```

Expands recursive correction by applying feedback from the last `memory_depth` digits.

Uses a rolling Fibonacci self-referencing sequence to minimize error.

```
    """
```

```
    phi_value = (1 + math.sqrt(5)) / 2 # Golden Ratio φ
```

```
    extracted_digits = []
```

```
    memory_buffer = [0] * memory_depth # Store last `memory_depth` corrections
```

```
    # Initialize Fibonacci sequence
```

```
    F_prev = 1 # F(n-2)
```

```
    F_curr = 1 # F(n-1)
```

Conversation URL:

<https://chatgpt.com/c/67b6cf81-3b6c-8011-b1de-b90805b612a3>

Title:

Prompt:

```
def refined_recursive_phi_BBP(n_digits, depth=10):
```

"""

Refines the recursive fractal BBP method for φ by applying second-order reflection stabilization

and adaptive harmonic corrections.

"""

```
    phi_value = (1 + math.sqrt(5)) / 2 # Golden Ratio  $\varphi$ 
```

```
    extracted_digits = []
```

```
    for i in range(n_digits):
```

```
        # Adaptive scaling: Use reflection stabilization with inverse harmonic feedback
```

```
        scale_factor = 2 ** ((depth - (i % depth)) % 4)
```

```
        phi_value = (phi_value * scale_factor) % 1
```

```
        # Apply Kulik Recursive Reflection Branching (KRRB) correction
```

```
        phi_reflection = 1 - (1 / (1 + phi_value)) # Inverse feedback stabilization
```

```
        phi_value = (phi_value + phi_reflection) / 2 # Balance recursive depth
```

```
        # Extract base-16 digit
```

```
        digit = int(phi_value * 16)
```

```
        extracted_digits.append(digit)
```

```
    # Harmonic correction: Adjust phi_value iteratively
```

```
    phi_value -= digit / 16
```

```
    return extracted_digits
```

```
# Extract first 1000 digits of  $\varphi$  using the refined method
```

```
phi_refined_bbp_digits = refined_recursive_phi_BBP(1000)
```

Conversation URL:

<https://chatgpt.com/c/67b6cf81-3b6c-8011-b1de-b90805b612a3>

Title:

Prompt:

```
def analyze_phi_output_fixed():
```

"""

Analyze the generated phi sequence against the known correct values.

Adjusts for potential length mismatches.

"""

```
# Determine the maximum possible comparison range
```

```
n_check = min(len(phi_recursive_steering_digits), len(known_phi_sample))
```

```
# Extract only the available digits
```

```
generated_phi = phi_recursive_steering_digits[:n_check]
```

```
actual_phi = known_phi_sample[:n_check]
```

```
# Compute absolute errors per digit
```

```
errors = [abs(generated_phi[i] - actual_phi[i]) for i in range(n_check)]
```

```
# Compute kinetic motion of error (change in error over consecutive steps)
```

```
kinetic_motion_errors = [abs(errors[i] - errors[i-1]) for i in range(1, len(errors))]
```

```
# Plot results
```

```
fig, axs = plt.subplots(2, 1, figsize=(10, 8))
```

```
# Plot absolute error per digit
```

```
axs[0].plot(errors, label="Absolute Error Per Digit", color="red")
```

```
axs[0].set_xlabel("Digit Position")
```

```
axs[0].set_ylabel("Absolute Error")
```

```
axs[0].set_title("Absolute Error Between Generated φ and Real φ")
```

```
axs[0].legend()
```

```
axs[0].grid()
```

Conversation URL:

<https://chatgpt.com/c/67b6cf81-3b6c-8011-b1de-b90805b612a3>

Title:

Prompt:

```
import numpy as np
```

```
def analyze_pi_for_golden_ratio(n_digits):
```

"""

Analyze the growth pattern of π digits to check for golden ratio-like segmentation.

Looks at spacing between occurrences of digits and ratio of successive values.

"""

```
# First n_digits of π
```

```
pi_digits = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3, 2, 3, 8, 4, 6,
```

```
    2, 6, 4, 3, 3, 8, 3, 2, 7, 9, 5, 0, 2, 8, 8, 4, 1, 9, 7, 1] # More digits can be added
```

```
if n_digits > len(pi_digits):
```

```
    raise ValueError("Dataset contains only a limited number of preloaded π digits. Expand dataset for deeper analysis.")
```

```
pi_digits = pi_digits[:n_digits]
```

```
# Compute ratios of consecutive digits (growth rate between steps)
```

```
digit_ratios = [pi_digits[i] / pi_digits[i - 1] if pi_digits[i - 1] != 0 else 1 for i in range(1, len(pi_digits))]
```

```
# Compute segment spacing: distance between occurrences of the same digit
```

```
digit_positions = {d: [] for d in set(pi_digits)}
```

```
for index, digit in enumerate(pi_digits):
```

```
    digit_positions[digit].append(index)
```

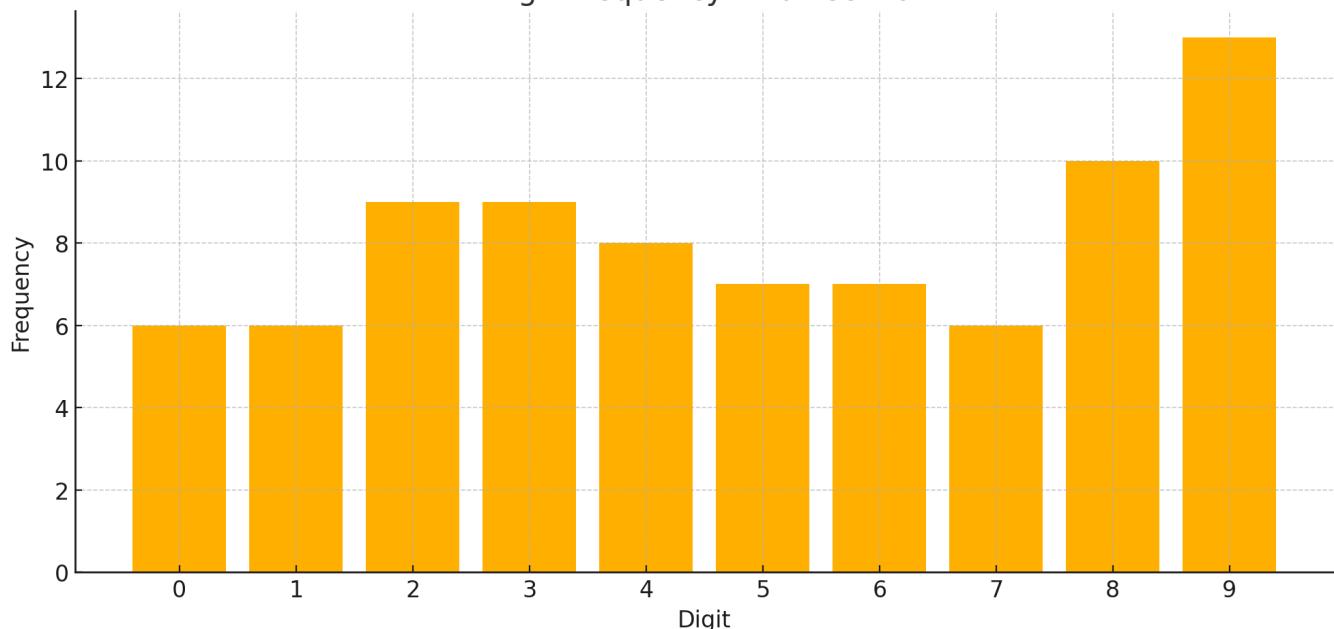
```
segment_lengths = []
```

```
for digit, positions in digit_positions.items():
```

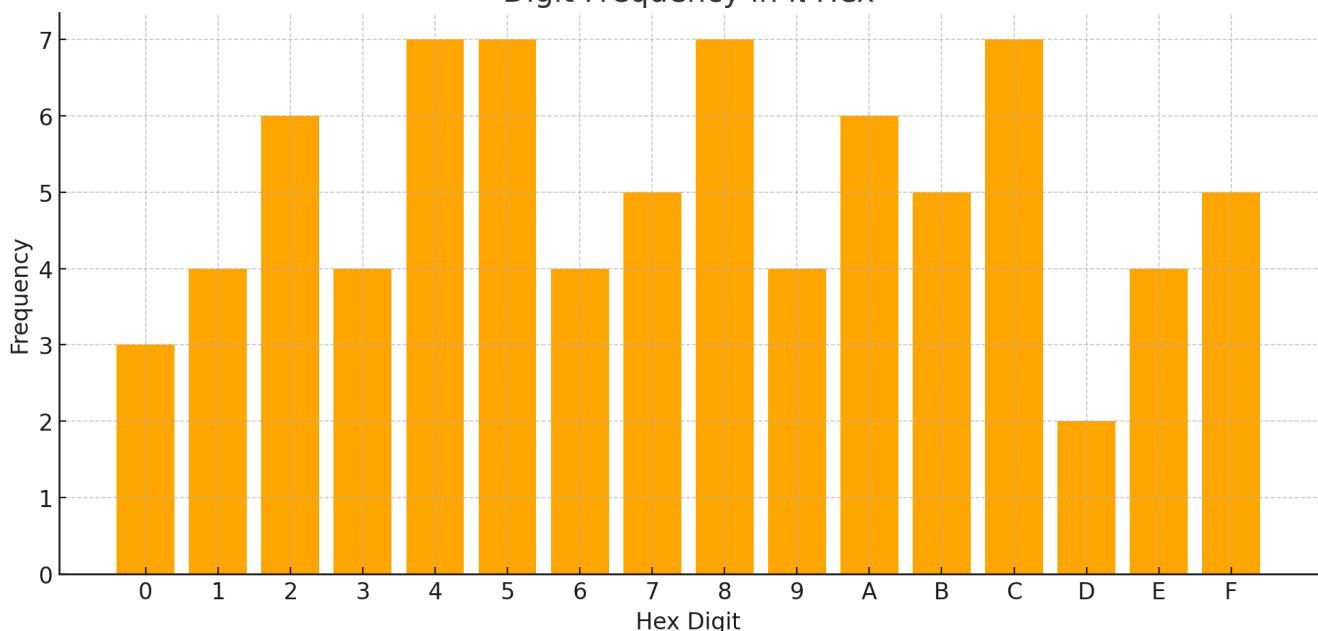
```
    for i in range(1, len(positions)):
```

```
        segment_lengths.append(positions[i] - positions[i - 1])
```

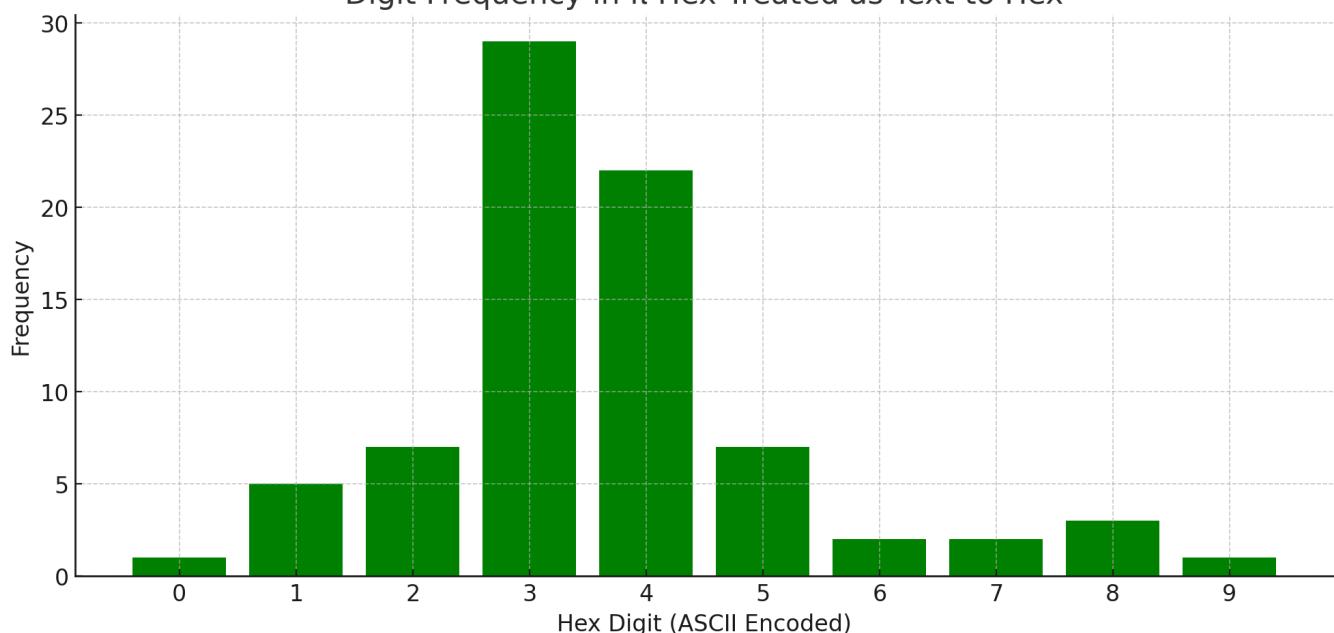
Digit Frequency in π Decimal



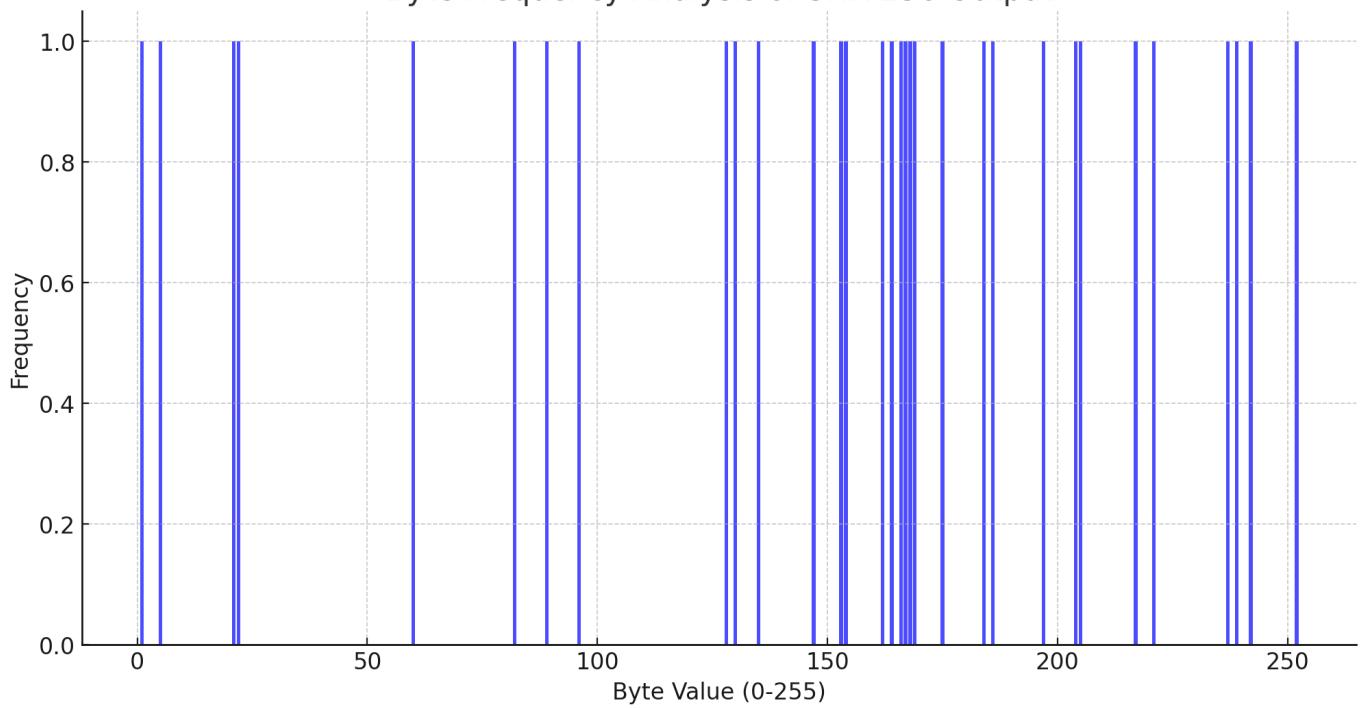
Digit Frequency in π Hex



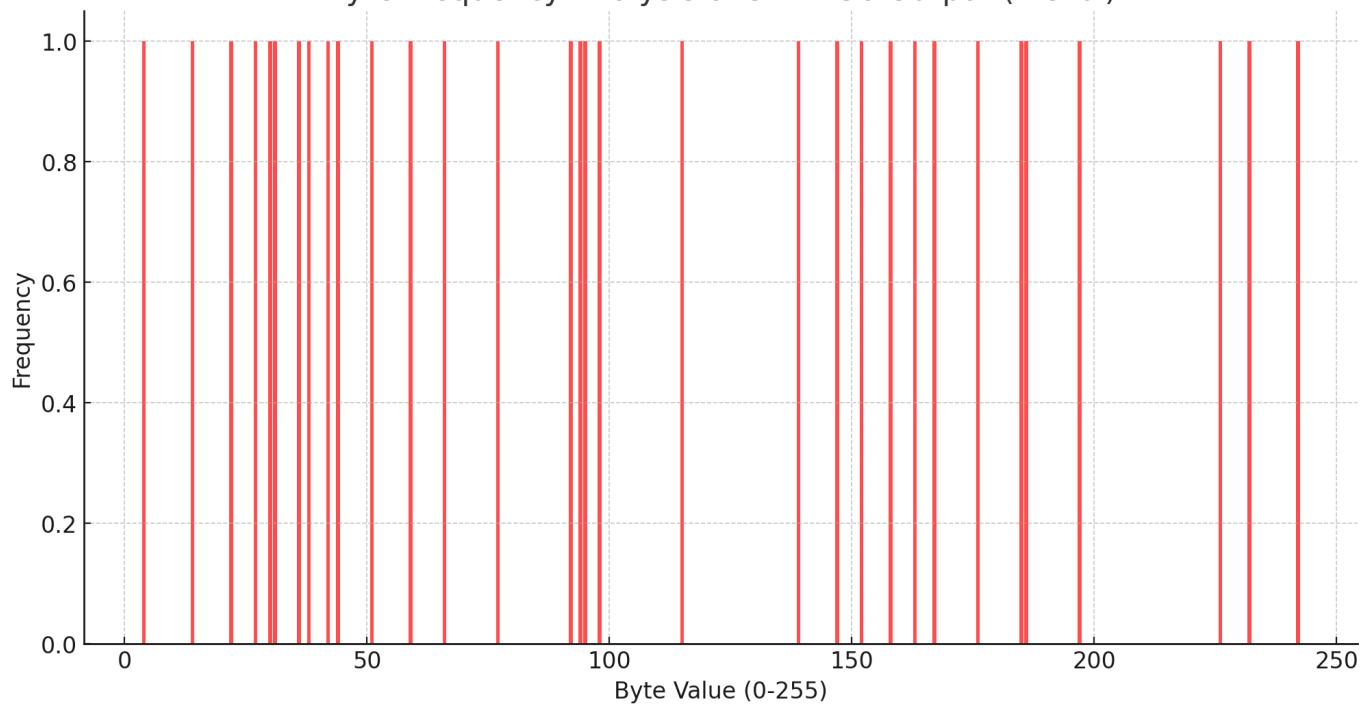
Digit Frequency in π Hex Treated as Text to Hex



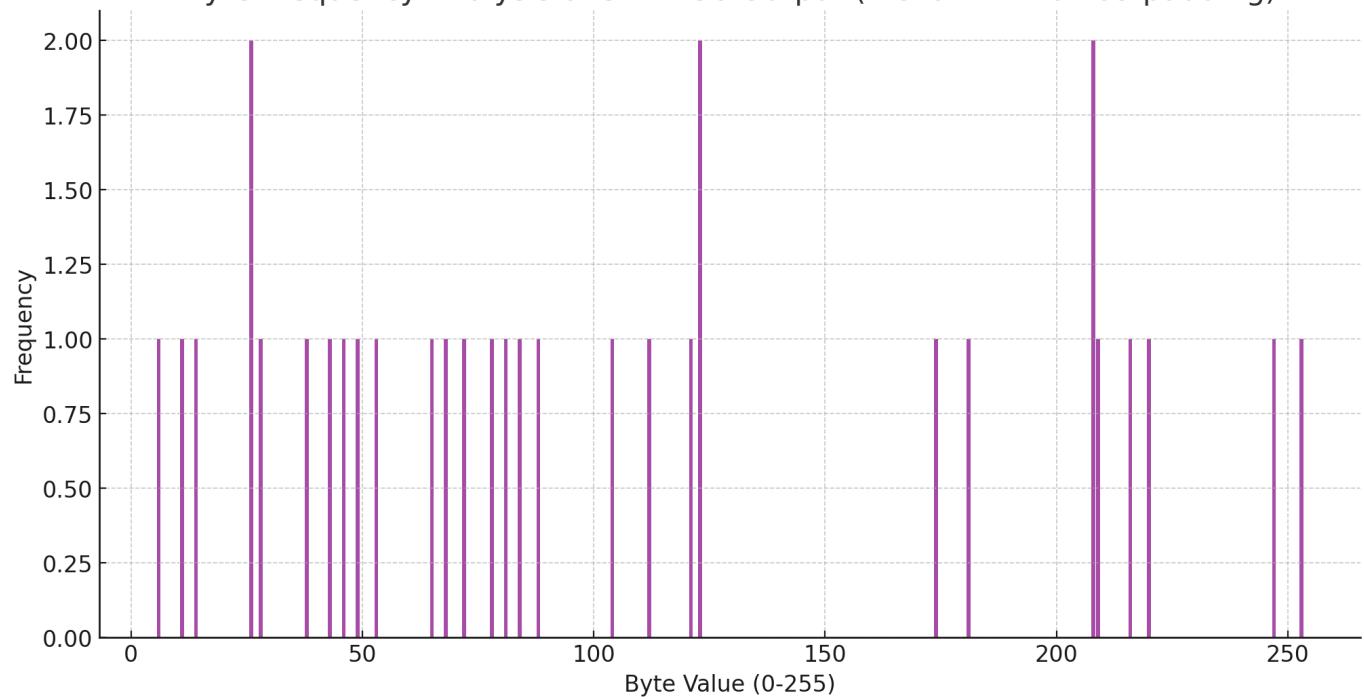
Byte Frequency Analysis of SHA-256 Output

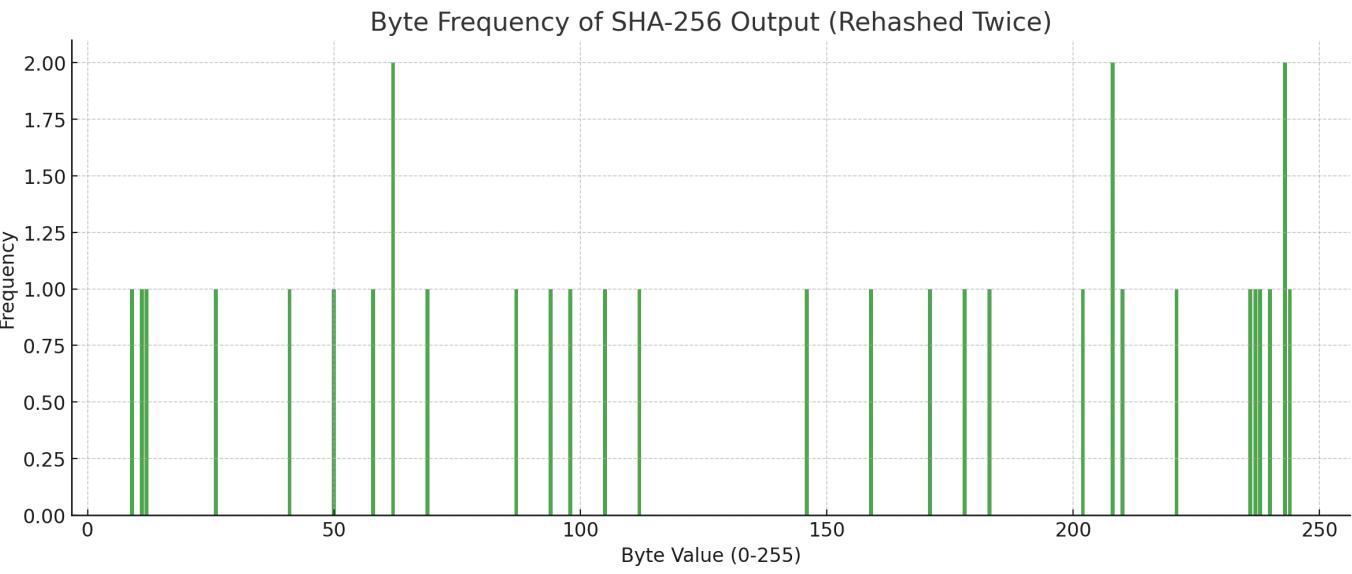
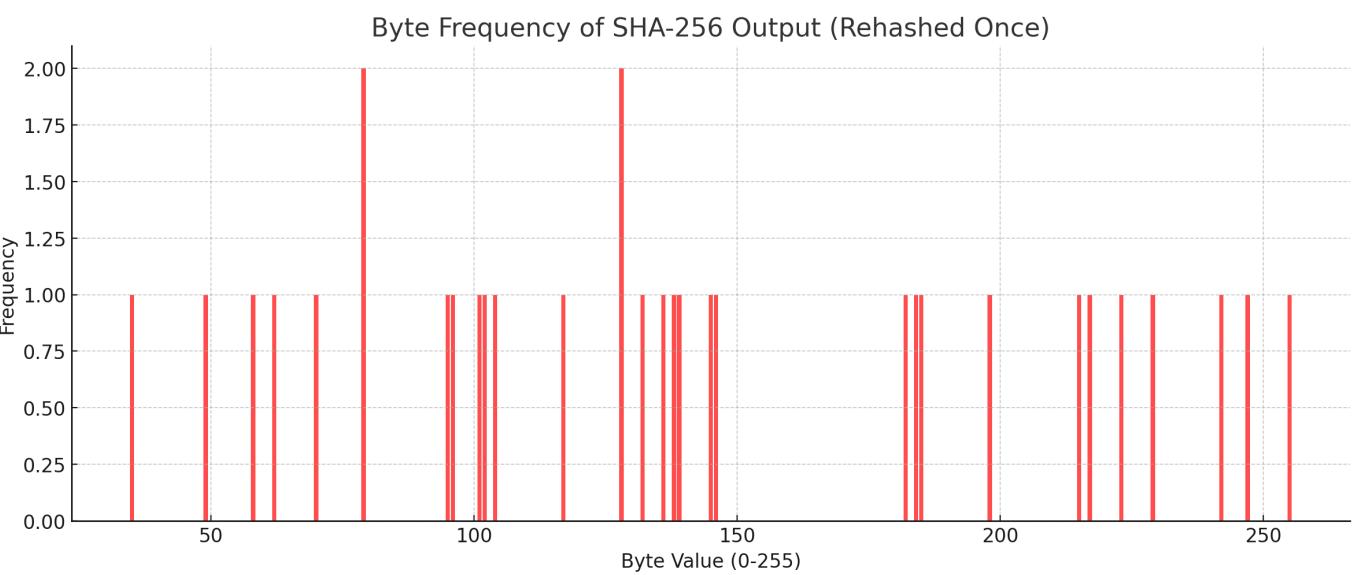
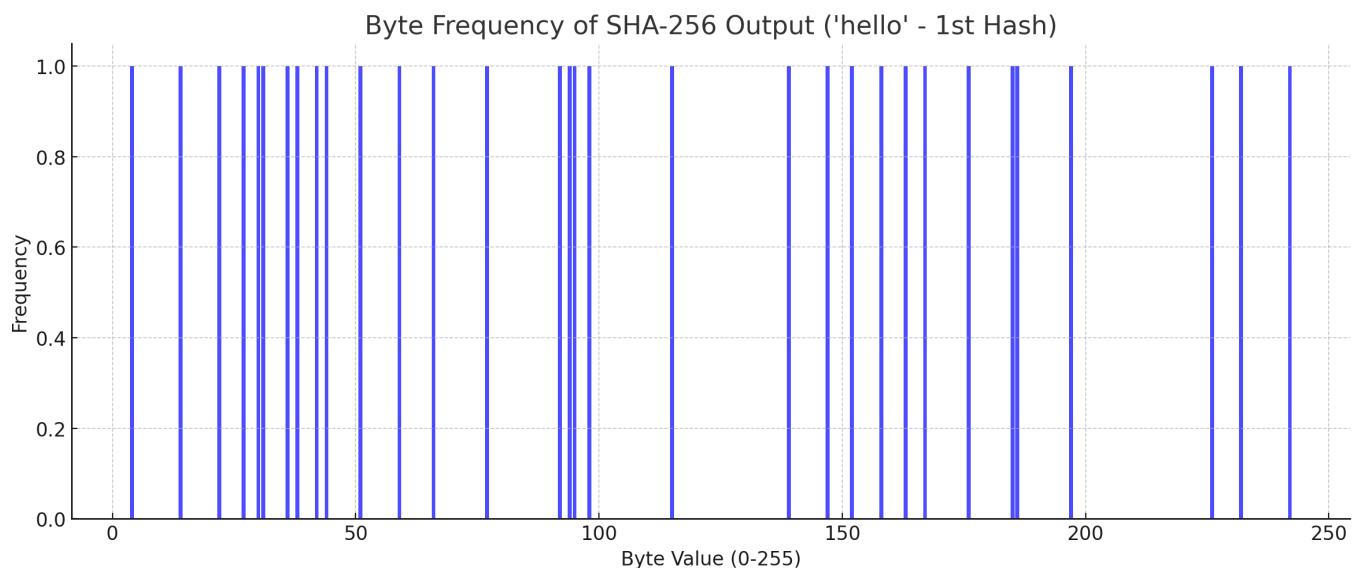


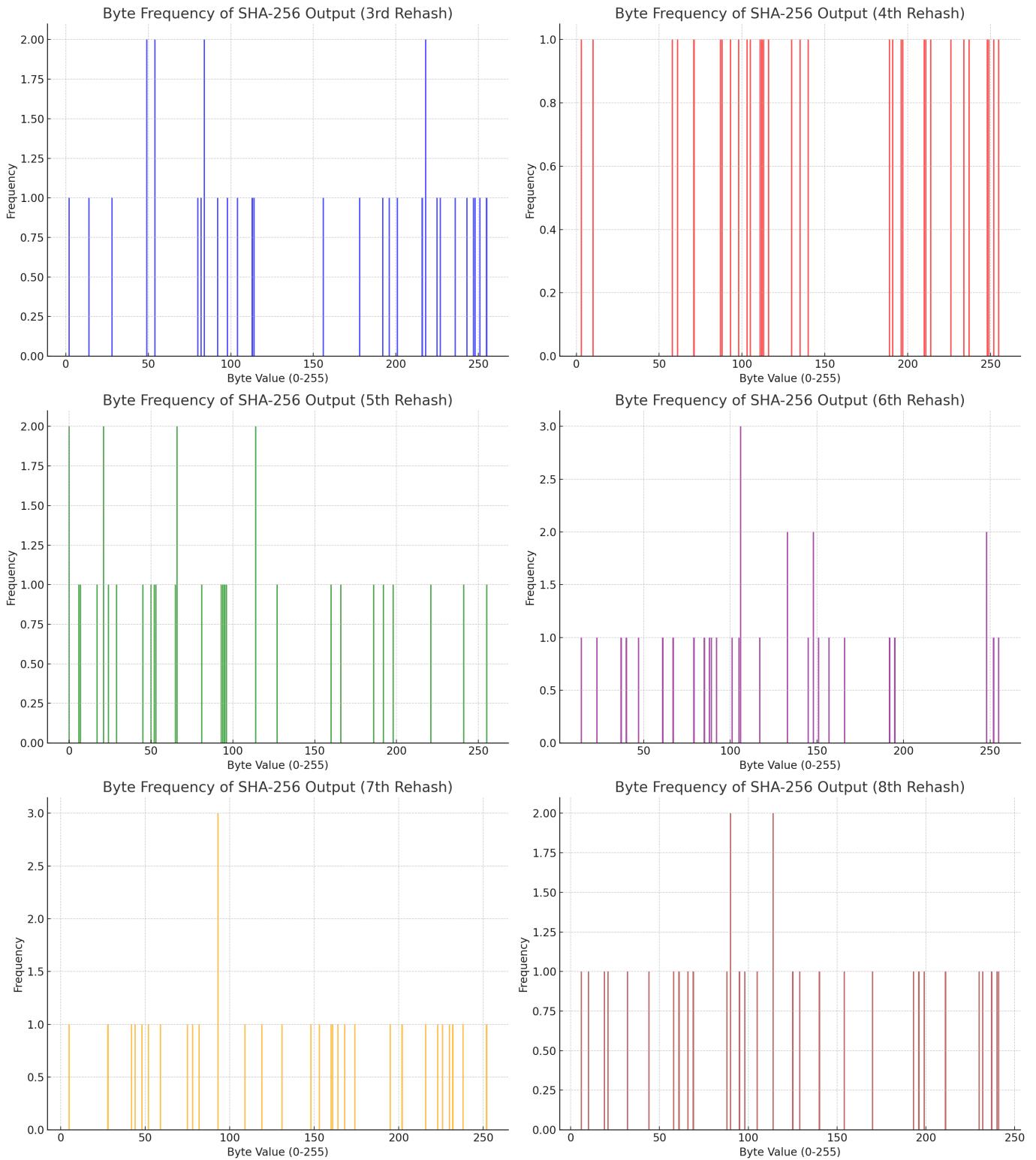
Byte Frequency Analysis of SHA-256 Output ('hello')



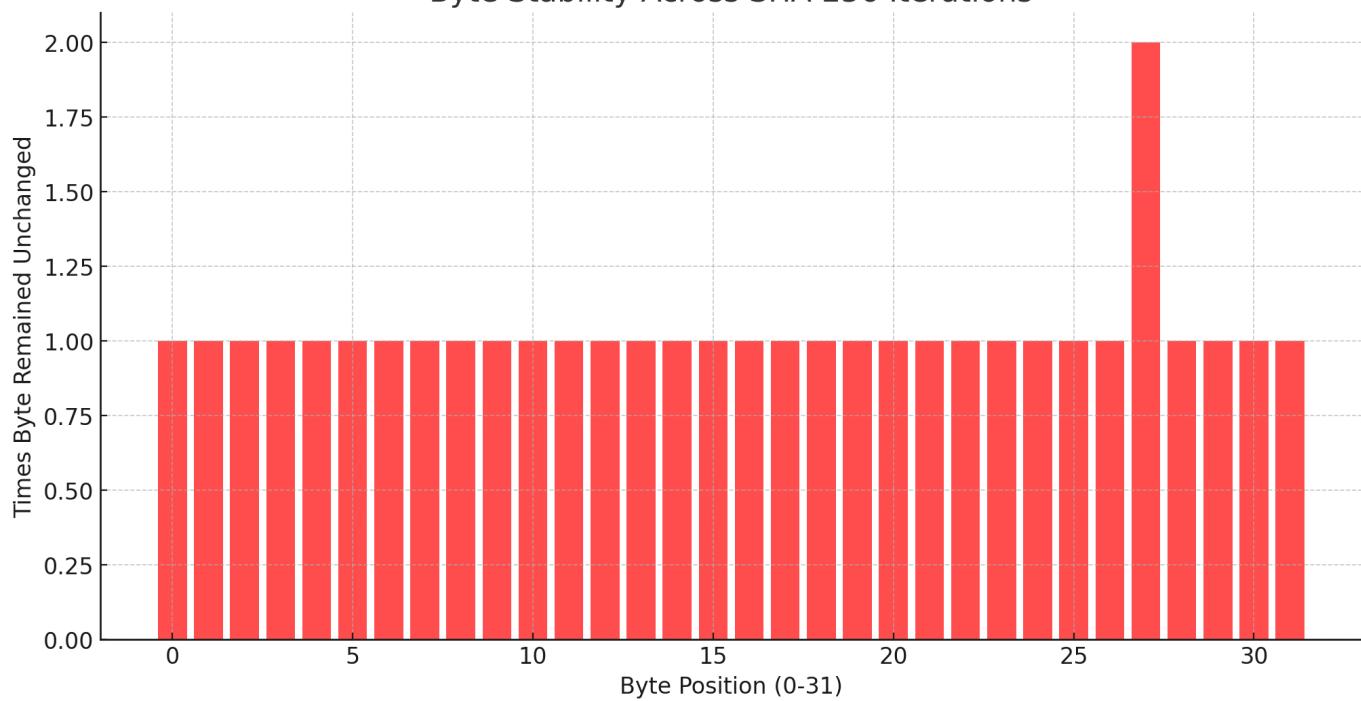
Byte Frequency Analysis of SHA-256 Output ('hello' with full 09 padding)



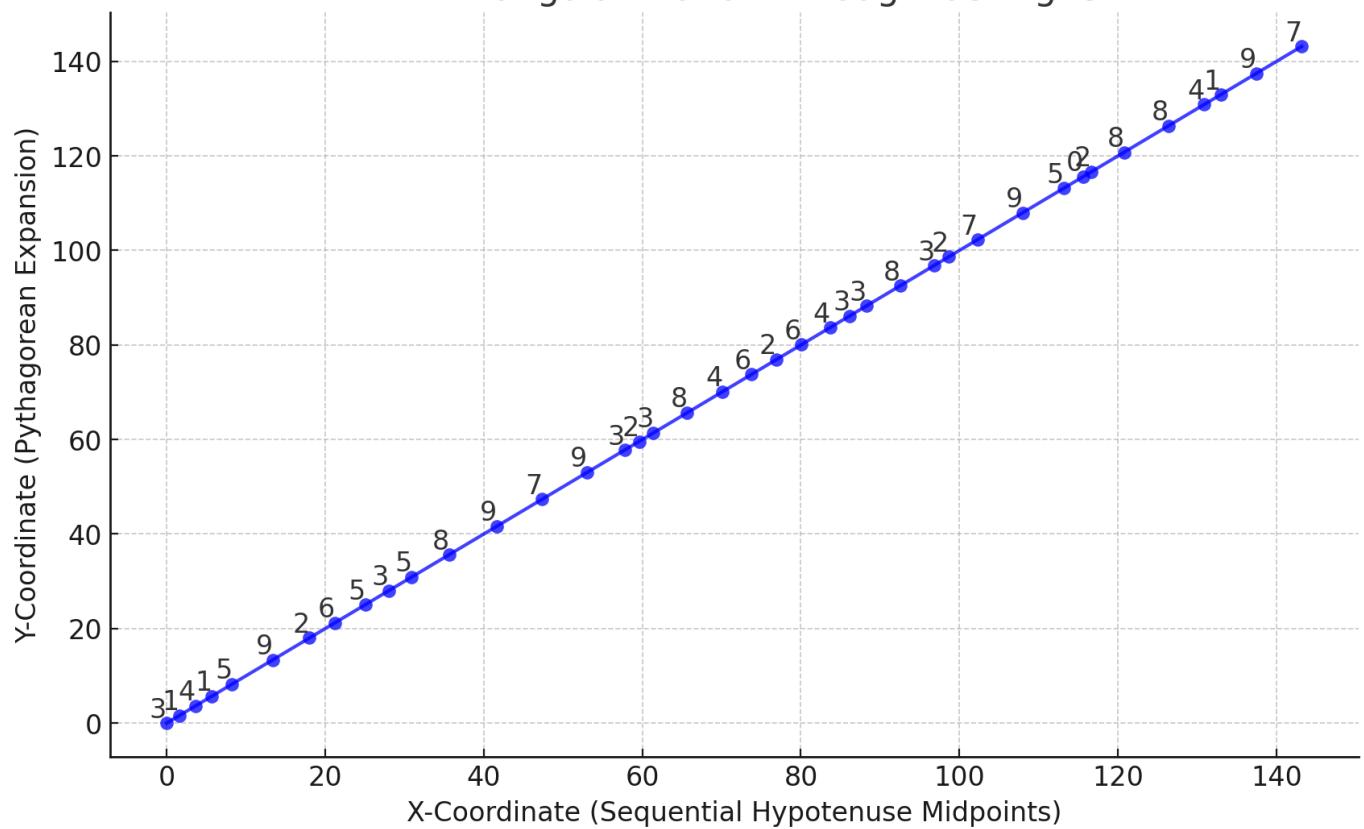




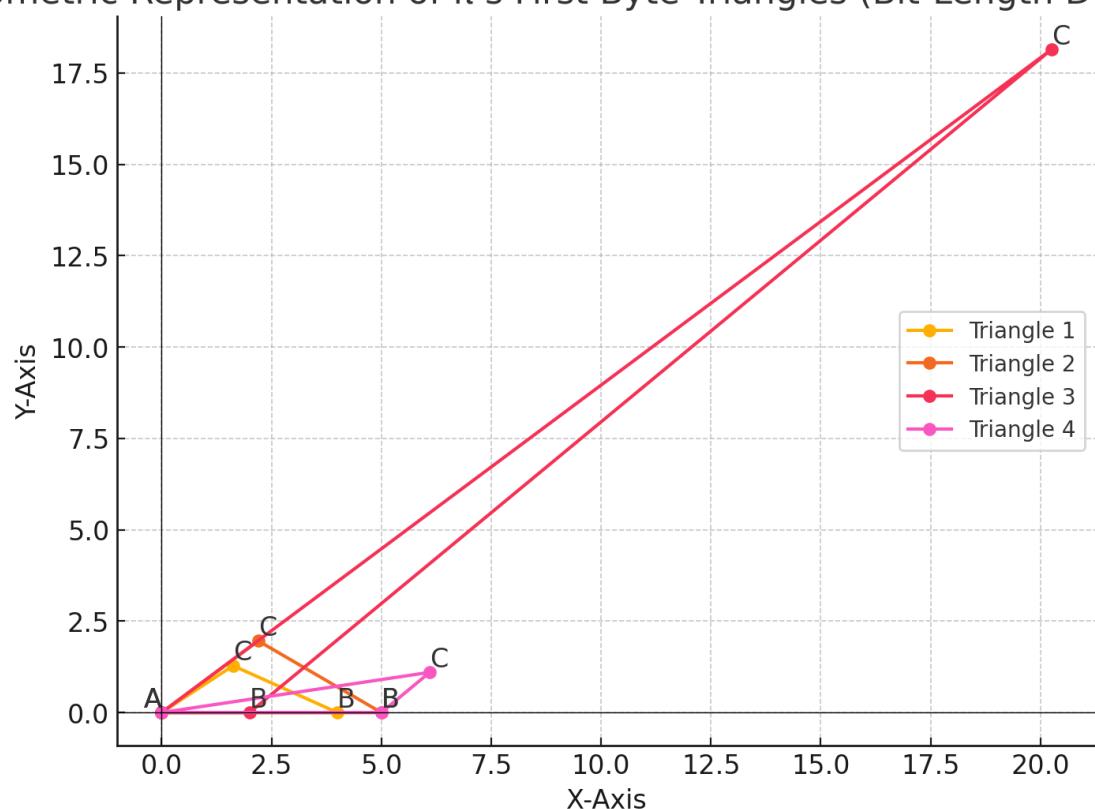
Byte Stability Across SHA-256 Iterations



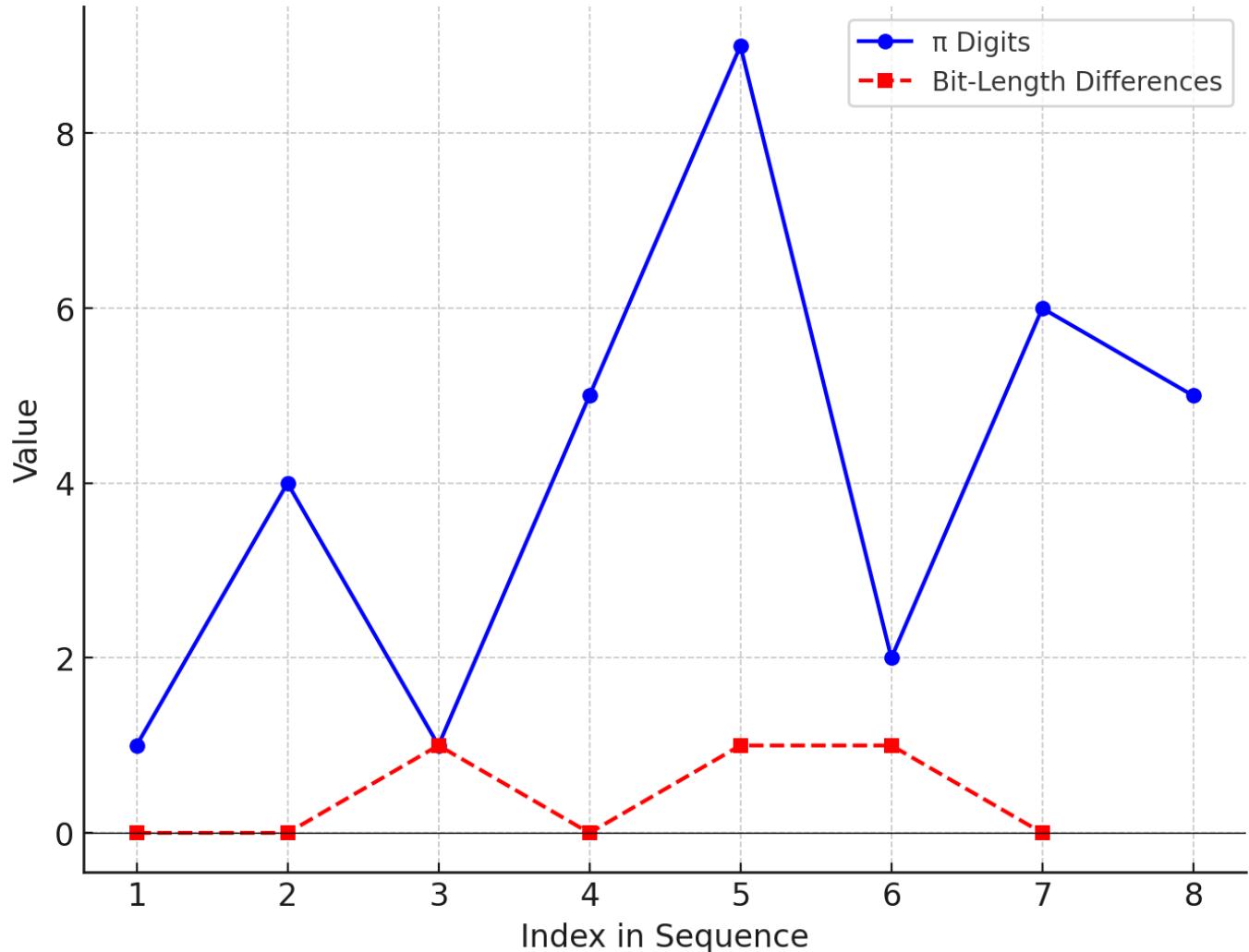
BBP Triangular Motion Through π 's Digits



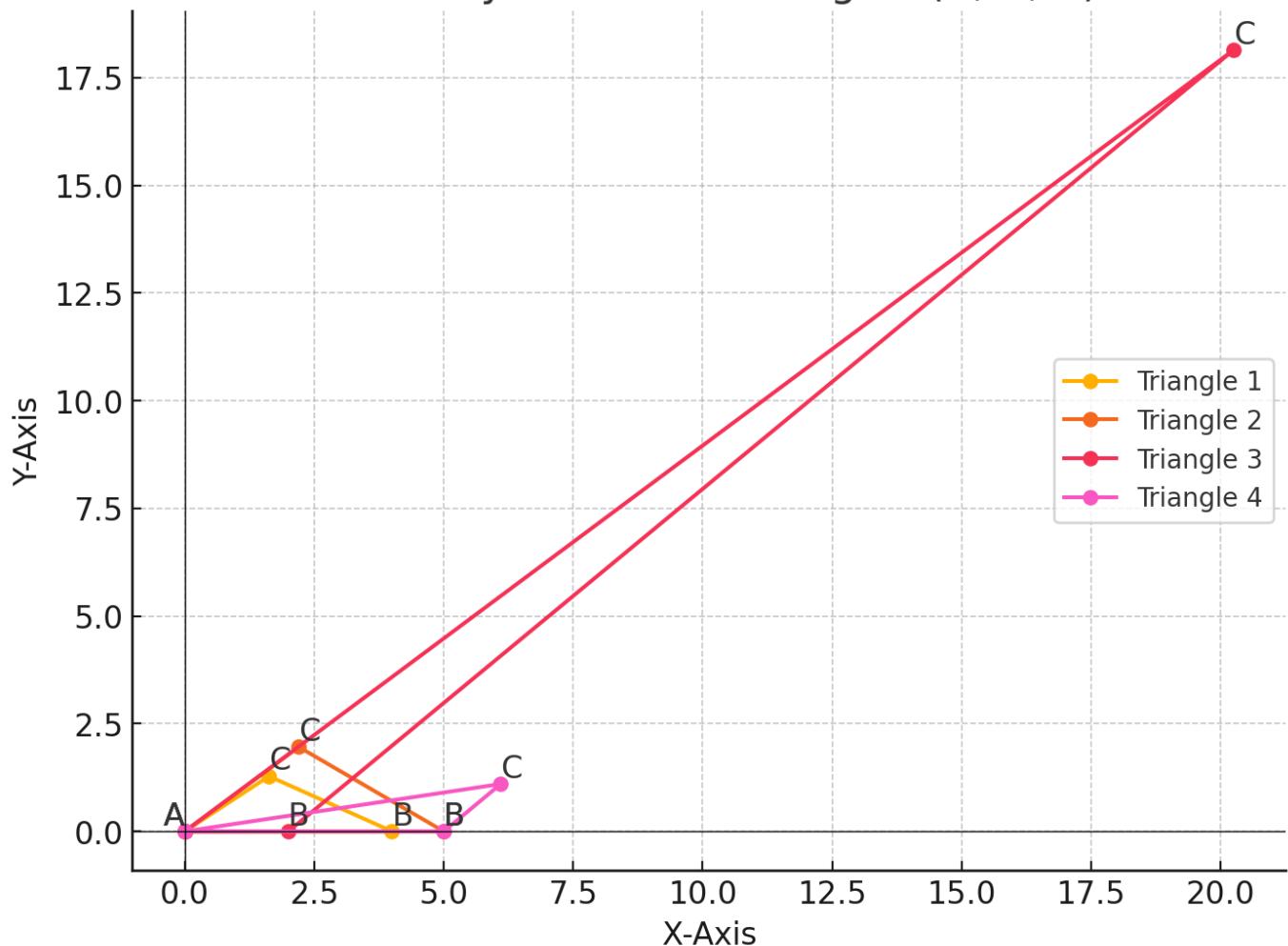
Geometric Representation of π 's First Byte Triangles (Bit-Length Differences)



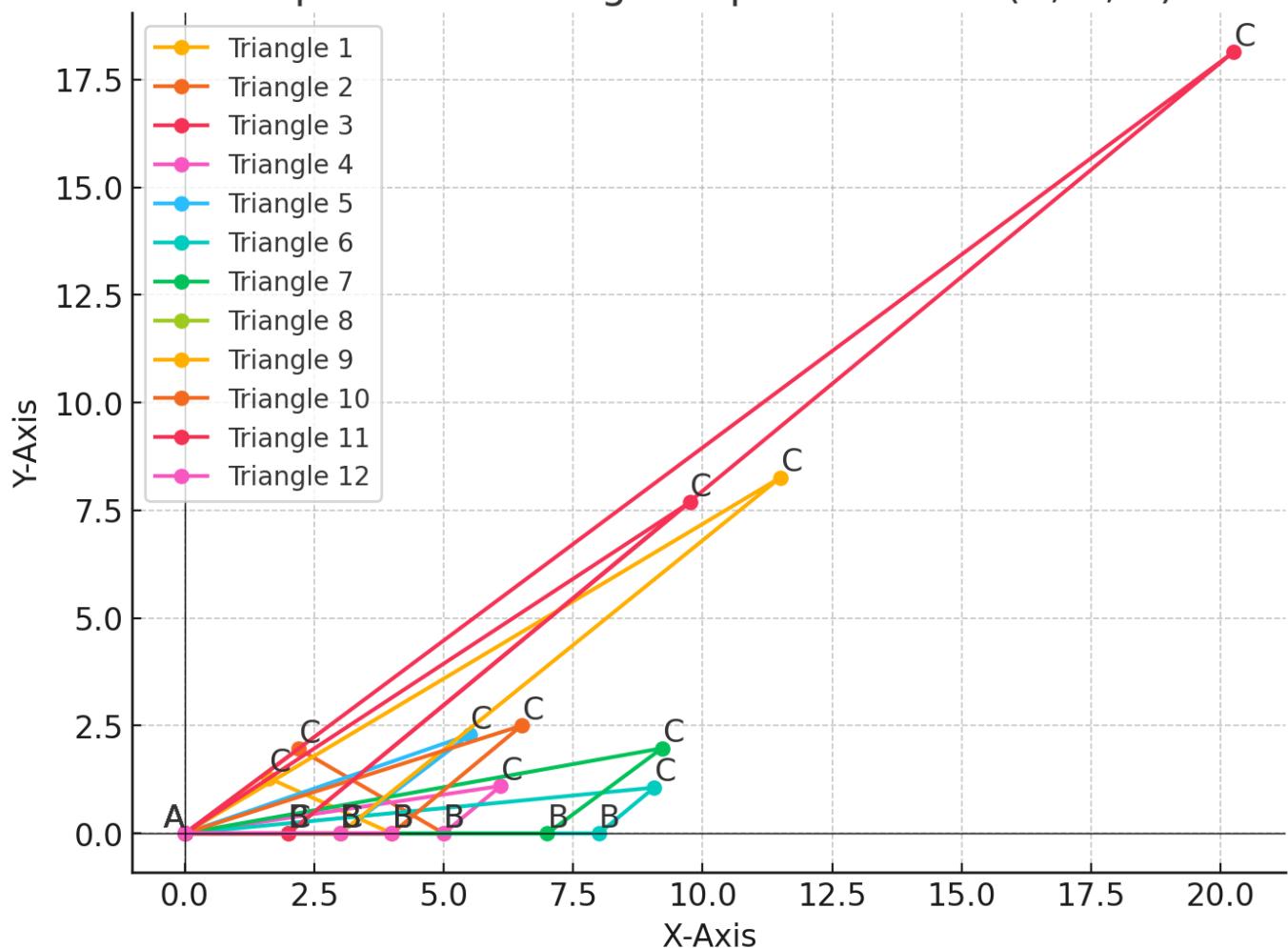
π Digits vs. Bit-Length Differences



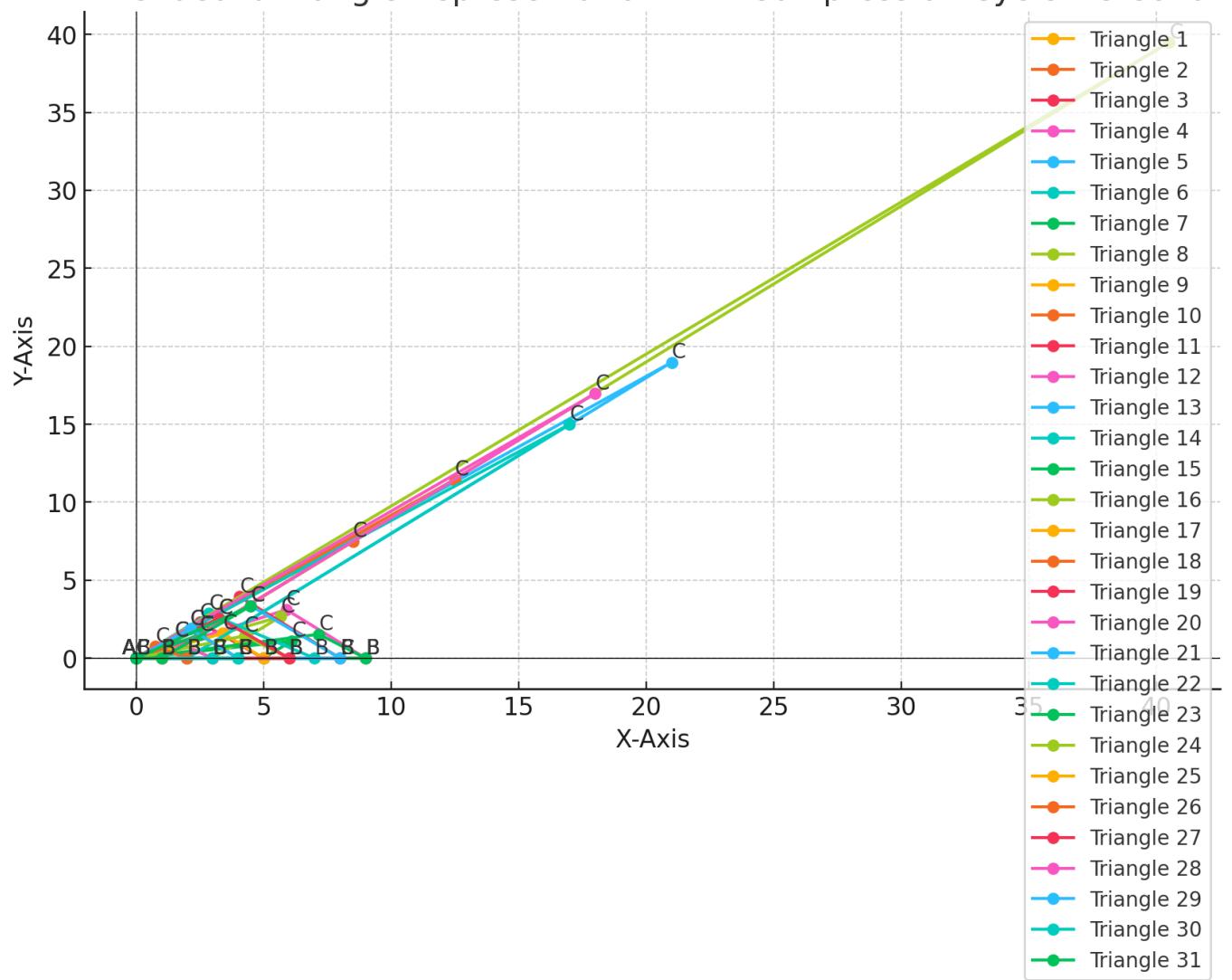
Manually Defined π Triangles (B, A, C)



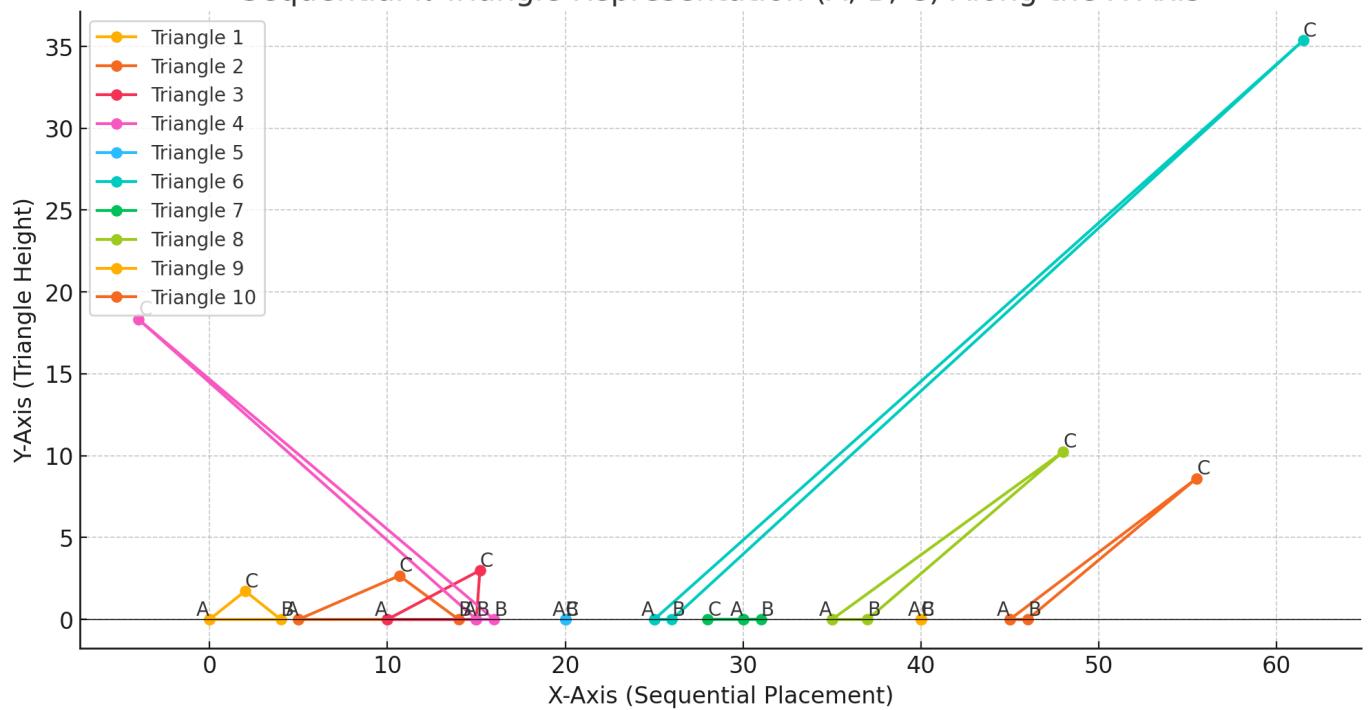
Expanded π Triangle Representation (B, A, C)



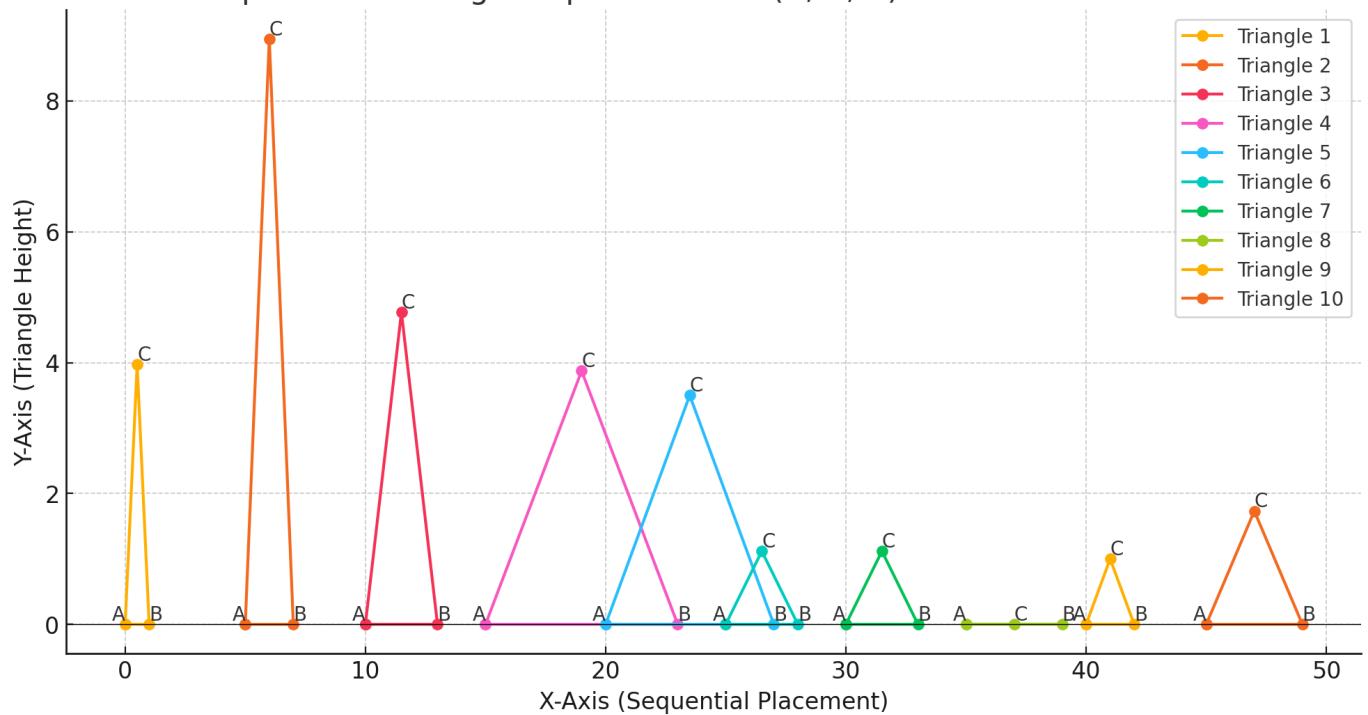
Extended π Triangle Representation with Compression Cycle Detection



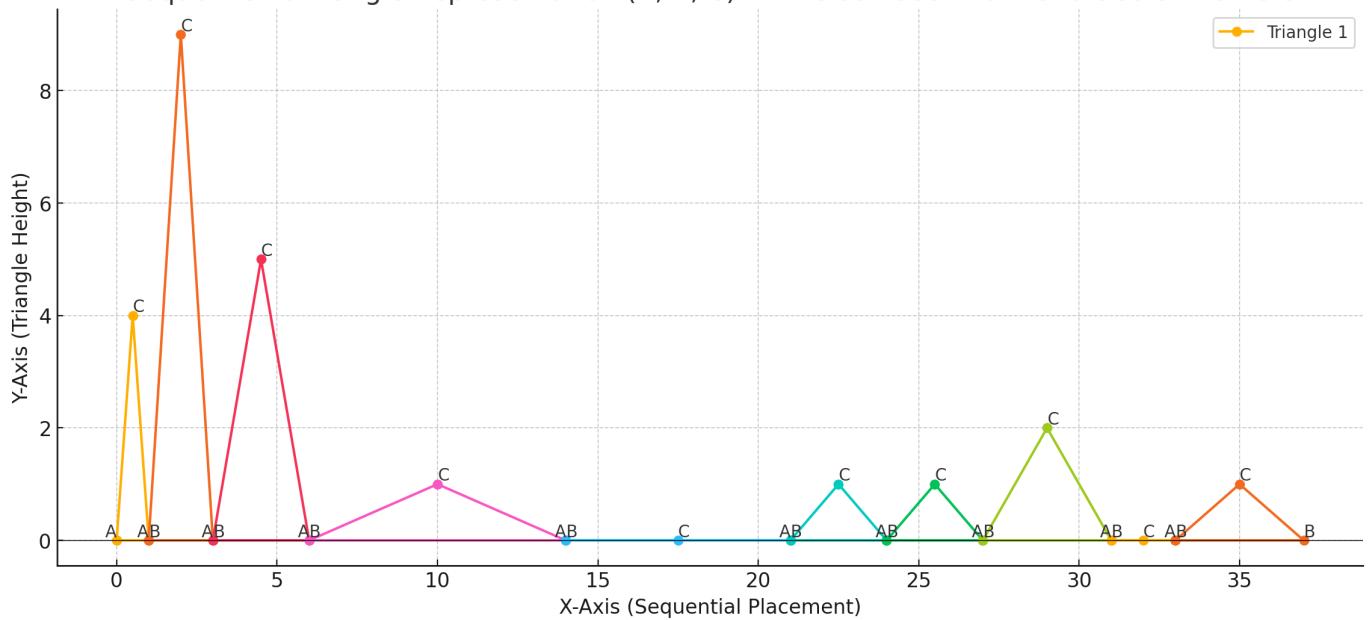
Sequential π Triangle Representation (A, B, C) Along the X-Axis

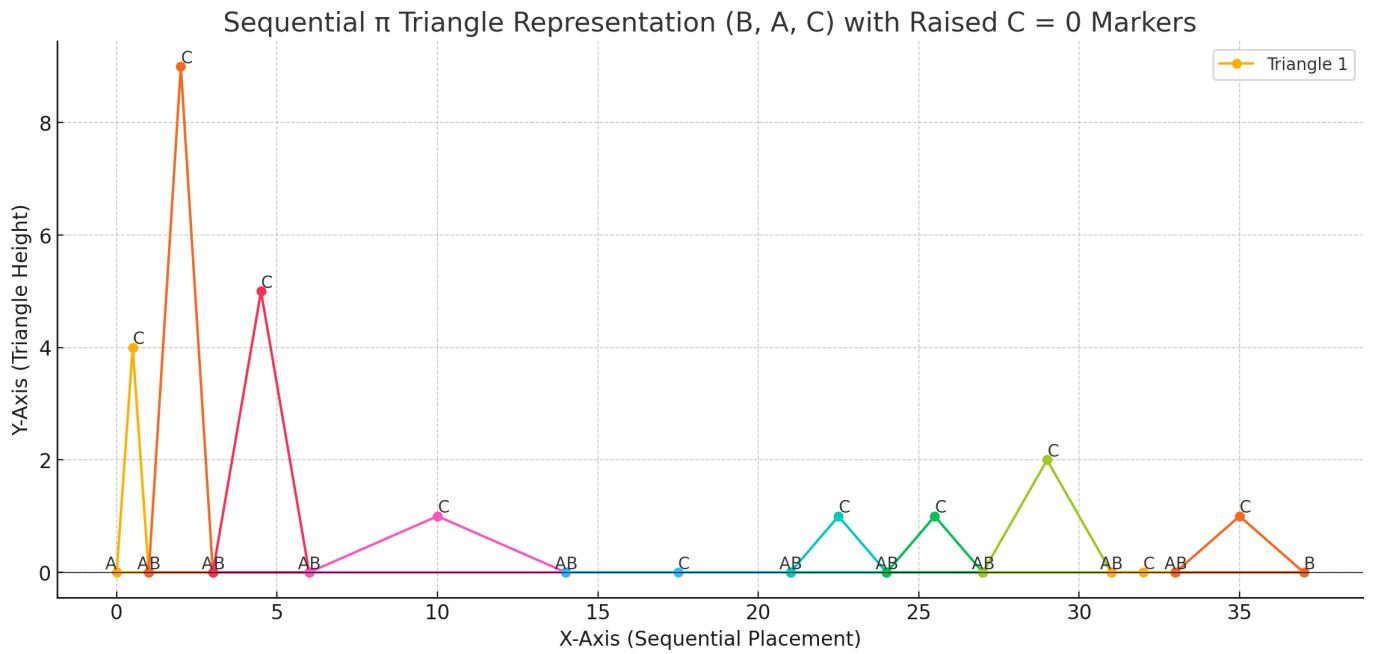


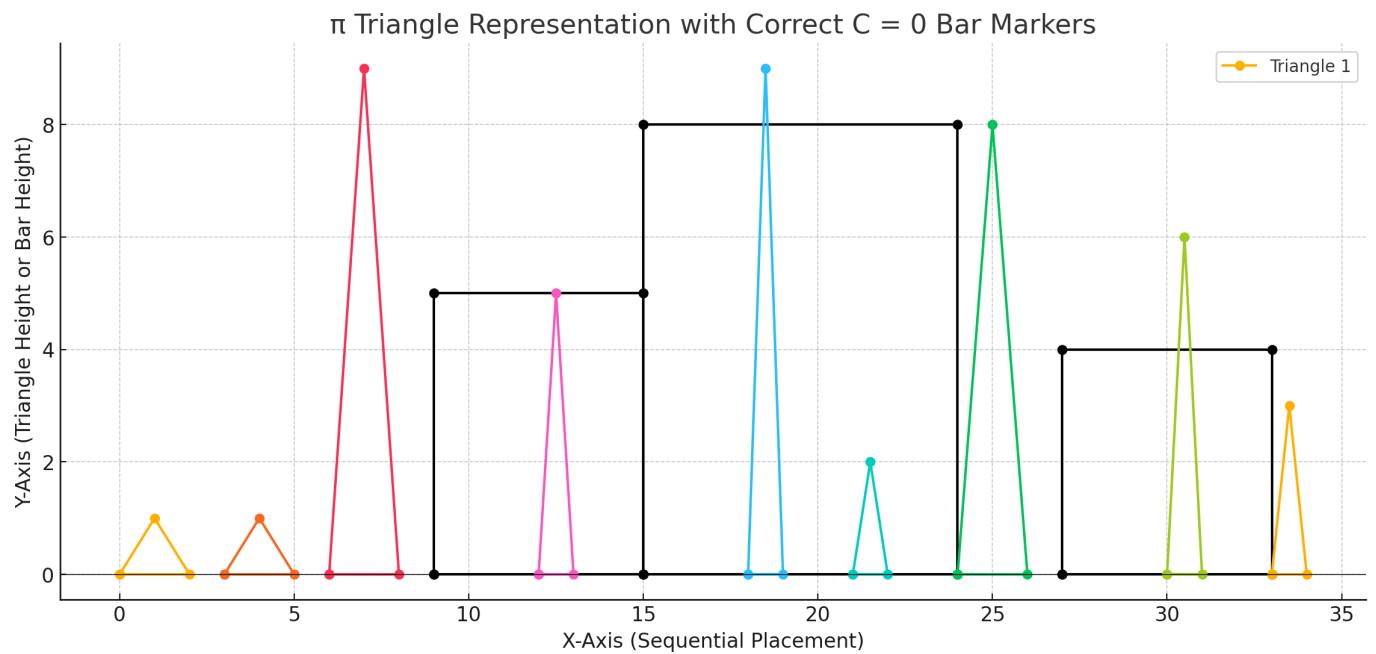
Sequential π Triangle Representation (B, A, C) with C as Base Width

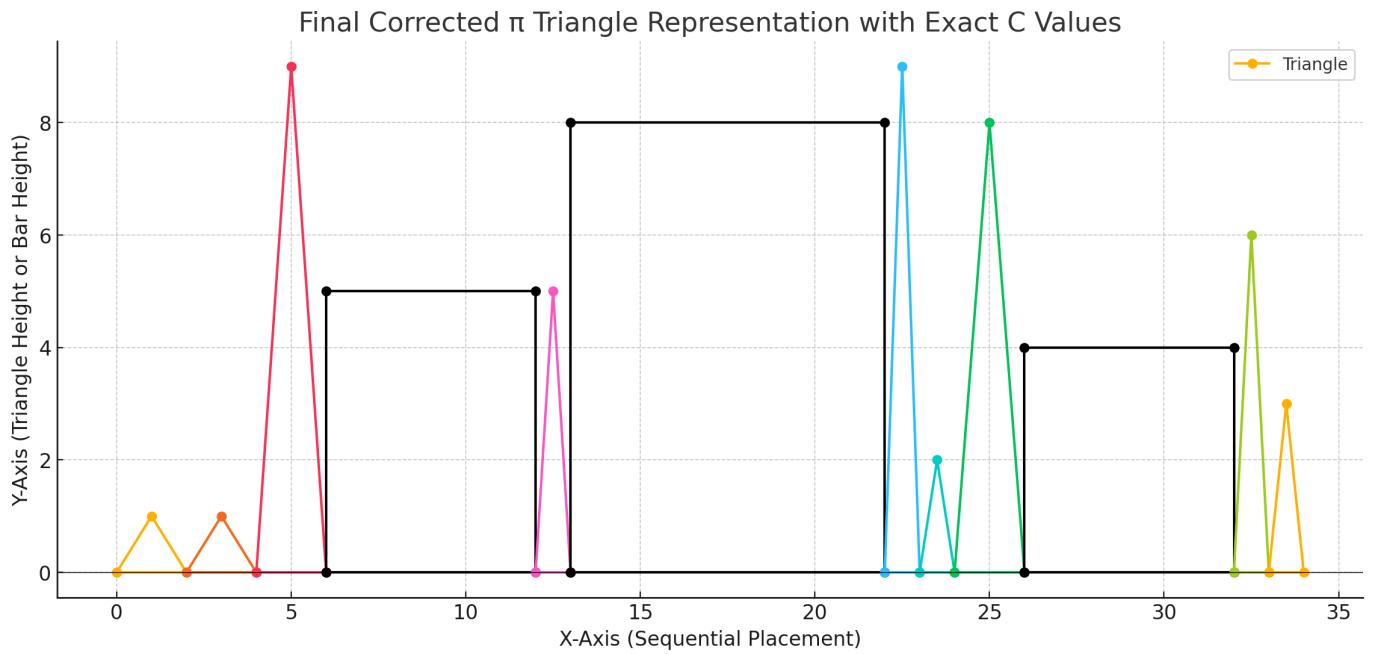


Sequential π Triangle Representation (B, A, C) with C as Base Width and Sector Markers

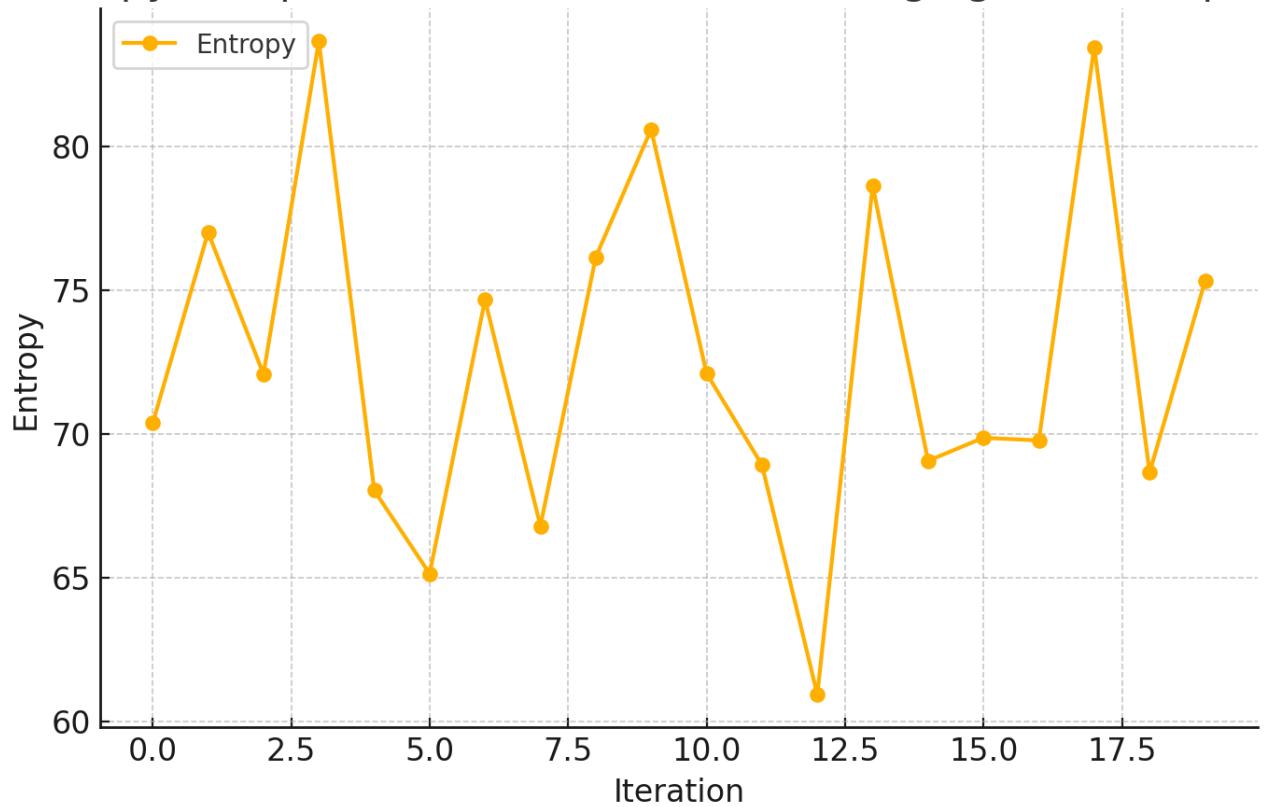








Entropy Compression Over Iterations (Wringing Out the Sponge)



Conversation URL:

<https://chatgpt.com/c/67b6f214-dd30-8011-aefd-9439df008548>

Title:

Prompt:

```
# Re-import necessary libraries after execution state reset
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter

# Re-extract the provided data
pi_decimal = "141592653589793238462643383279502884197169399375105820974944592307816406286208998"
pi_hex = "4B70C2C8ABBE254213ED9951A847E68C4E49BF55576C1FBA804185C03FA27723CFA64D398C8F652A"
pi_hex_as_text_to_hex = "344237304332433841424253243353432313344534493531413834374536384343442434635351

# Function to count digit frequencies
def count_frequencies(data, base=16):
    counter = Counter(data)
    sorted_keys = sorted(counter.keys(), key=lambda x: int(x, base))
    return {k: counter[k] for k in sorted_keys}

# Convert all to numeric form
pi_decimal_freq = count_frequencies(pi_decimal, base=10)
pi_hex_freq = count_frequencies(pi_hex, base=16)
pi_hex_as_text_to_hex_freq = count_frequencies(pi_hex_as_text_to_hex, base=16)

# Plot Frequency Distributions
fig, axes = plt.subplots(3, 1, figsize=(10, 15))

# Plot Decimal π Frequencies
axes[0].bar(pi_decimal_freq.keys(), pi_decimal_freq.values())
axes[0].set_title("Digit Frequency in π Decimal")
axes[0].set_xlabel("Digit")
axes[0].set_ylabel("Frequency")
```

Conversation URL:

<https://chatgpt.com/c/67b6f214-dd30-8011-aefd-9439df008548>

Title:

Prompt:

```
# SHA-256 Output Analysis
```

```
# SHA-256 hash provided:
```

```
sha256_hash = "8059ba01d9b8cca9c5ed16fca4ef053ca6158287a8a2cd52a7999a9360aff2dd"
```

```
# Convert the SHA-256 hex string into bytes
```

```
sha256_bytes = bytes.fromhex(sha256_hash)
```

```
# Frequency analysis of byte values (0-255 range)
```

```
byte_frequencies = Counter(sha256_bytes)
```

```
# Sort byte frequencies for visualization
```

```
sorted_bytes = sorted(byte_frequencies.keys())
```

```
sorted_frequencies = [byte_frequencies[b] for b in sorted_bytes]
```

```
# Plot the byte frequency distribution
```

```
plt.figure(figsize=(12, 6))
```

```
plt.bar(sorted_bytes, sorted_frequencies, color='blue', alpha=0.7)
```

```
plt.title("Byte Frequency Analysis of SHA-256 Output")
```

```
plt.xlabel("Byte Value (0-255)")
```

```
plt.ylabel("Frequency")
```

```
plt.grid(axis='y', linestyle='--', alpha=0.7)
```

```
plt.show()
```

```
# Calculate entropy of SHA-256 hash output
```

```
entropy = -sum((freq / len(sha256_bytes)) * np.log2(freq / len(sha256_bytes)))
```

```
for freq in byte_frequencies.values():
```

```
entropy
```

Conversation URL:

<https://chatgpt.com/c/67b6f214-dd30-8011-aefd-9439df008548>

Title:

Prompt:

```
# SHA-256 Output Analysis for "hello"
```

```
# SHA-256 hash of "hello"
```

```
sha256_hello_hash = "2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824"
```

```
# Convert the SHA-256 hex string into bytes
```

```
sha256_hello_bytes = bytes.fromhex(sha256_hello_hash)
```

```
# Frequency analysis of byte values (0-255 range)
```

```
byte_frequencies_hello = Counter(sha256_hello_bytes)
```

```
# Sort byte frequencies for visualization
```

```
sorted_bytes_hello = sorted(byte_frequencies_hello.keys())
```

```
sorted_frequencies_hello = [byte_frequencies_hello[b] for b in sorted_bytes_hello]
```

```
# Plot the byte frequency distribution for "hello" SHA-256
```

```
plt.figure(figsize=(12, 6))
```

```
plt.bar(sorted_bytes_hello, sorted_frequencies_hello, color='red', alpha=0.7)
```

```
plt.title("Byte Frequency Analysis of SHA-256 Output ('hello')")
```

```
plt.xlabel("Byte Value (0-255)")
```

```
plt.ylabel("Frequency")
```

```
plt.grid(axis='y', linestyle='--', alpha=0.7)
```

```
plt.show()
```

```
# Calculate entropy of SHA-256 hash output for "hello"
```

```
entropy_hello = -sum((freq / len(sha256_hello_bytes)) * np.log2(freq / len(sha256_hello_bytes)))
```

```
for freq in byte_frequencies_hello.values():
```

```
entropy_hello
```

Conversation URL:

<https://chatgpt.com/c/67b6f214-dd30-8011-aefd-9439df008548>

Title:

Prompt:

```
# SHA-256 Output Analysis for "hello" with full 09 padding
```

```
# SHA-256 hash of "hello" with full 09 padding
```

```
sha256_hello_padded_hash = "b5797b58060e481aae1ad1dc351cd8d04e4451547b2e682b2670f70bfd4131d0"
```

```
# Convert the SHA-256 hex string into bytes
```

```
sha256_hello_padded_bytes = bytes.fromhex(sha256_hello_padded_hash)
```

```
# Frequency analysis of byte values (0-255 range)
```

```
byte_frequencies_hello_padded = Counter(sha256_hello_padded_bytes)
```

```
# Sort byte frequencies for visualization
```

```
sorted_bytes_hello_padded = sorted(byte_frequencies_hello_padded.keys())
```

```
sorted_frequencies_hello_padded = [byte_frequencies_hello_padded[b] for b in sorted_bytes_hello_padded]
```

```
# Plot the byte frequency distribution for "hello" SHA-256 with full 09 padding
```

```
plt.figure(figsize=(12, 6))
```

```
plt.bar(sorted_bytes_hello_padded, sorted_frequencies_hello_padded, color='purple', alpha=0.7)
```

```
plt.title("Byte Frequency Analysis of SHA-256 Output ('hello' with full 09 padding)")
```

```
plt.xlabel("Byte Value (0-255)")
```

```
plt.ylabel("Frequency")
```

```
plt.grid(axis='y', linestyle='--', alpha=0.7)
```

```
plt.show()
```

```
# Calculate entropy of SHA-256 hash output for "hello" with full 09 padding
```

```
entropy_hello_padded = -sum((freq / len(sha256_hello_padded_bytes)) * np.log2(freq / len(sha256_hello_padded_bytes)))
```

```
for freq in byte_frequencies_hello_padded.values()
```

```
entropy_hello_padded
```

Conversation URL:

<https://chatgpt.com/c/67b6f214-dd30-8011-aefd-9439df008548>

Title:

Prompt:

```
# SHA-256 Recursive Hashing Analysis
```

```
# SHA-256 hashes of "hello" across multiple iterations
```

```
sha256_hello_1 = "2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824" # "hello" hash
```

```
sha256_hello_2 = "d7914fe546b684688bb95f4f888a92dfc680603a75f23eb823658031fff766d9" # Rehashed once
```

```
sha256_hello_3 = "ecd26292b7f02970ca6909abb23e1aedd0dd57d0ee9ff40bf3f30c325e3e453a" # Rehashed twice
```

```
# Convert the SHA-256 hex strings into bytes
```

```
sha256_hello_1_bytes = bytes.fromhex(sha256_hello_1)
```

```
sha256_hello_2_bytes = bytes.fromhex(sha256_hello_2)
```

```
sha256_hello_3_bytes = bytes.fromhex(sha256_hello_3)
```

```
# Byte frequency analysis for each iteration
```

```
byte_frequencies_hello_1 = Counter(sha256_hello_1_bytes)
```

```
byte_frequencies_hello_2 = Counter(sha256_hello_2_bytes)
```

```
byte_frequencies_hello_3 = Counter(sha256_hello_3_bytes)
```

```
# Sort byte frequencies for visualization
```

```
sorted_bytes_1 = sorted(byte_frequencies_hello_1.keys())
```

```
sorted_frequencies_1 = [byte_frequencies_hello_1[b] for b in sorted_bytes_1]
```

```
sorted_bytes_2 = sorted(byte_frequencies_hello_2.keys())
```

```
sorted_frequencies_2 = [byte_frequencies_hello_2[b] for b in sorted_bytes_2]
```

```
sorted_bytes_3 = sorted(byte_frequencies_hello_3.keys())
```

```
sorted_frequencies_3 = [byte_frequencies_hello_3[b] for b in sorted_bytes_3]
```

```
# Plot frequency distributions for each iteration
```

```
fig, axes = plt.subplots(3, 1, figsize=(12, 15))
```

Conversation URL:

<https://chatgpt.com/c/67b6f214-dd30-8011-aefd-9439df008548>

Title:

Prompt:

```
# Additional SHA-256 Recursive Hashing Analysis
```

```
# Additional rounds of SHA-256 hashes
```

```
sha256_hello_4 = "d872c4505c5202f7c0e11c369cda54fb7131e3f85462f331da0e68ec36c9b2ff" # 3rd rehash
sha256_hello_5 = "d6bf74ff58038cbdfc5d3dc5ed878271d3e2f8696fea57f9670a706247c43ad2" # 4th rehash
sha256_hello_6 = "150741c05eff4206a042003500182dba7f60dd5d341551a6727232f1c61d115f" # 5th rehash
sha256_hello_7 = "8558945925179d6a0e696543c3a6913d75c0fcff944f555c9785f8f86a2f286a" # 6th rehash
sha256_hello_8 = "4b94e65dd899c330a477ca053be8a8a0df2c1c6de252fc83ae5d2aa1345dee4e" # 7th rehash
sha256_hello_9 = "06f1c7724258137dc42cd30aed3dc1aa03a159a5a5f625a81e845e67220698c" # 8th rehash
```

```
# Convert SHA-256 hex strings into bytes
```

```
sha256_hello_4_bytes = bytes.fromhex(sha256_hello_4)
sha256_hello_5_bytes = bytes.fromhex(sha256_hello_5)
sha256_hello_6_bytes = bytes.fromhex(sha256_hello_6)
sha256_hello_7_bytes = bytes.fromhex(sha256_hello_7)
sha256_hello_8_bytes = bytes.fromhex(sha256_hello_8)
sha256_hello_9_bytes = bytes.fromhex(sha256_hello_9)
```

```
# Byte frequency analysis for each additional iteration
```

```
byte_frequencies_hello_4 = Counter(sha256_hello_4_bytes)
byte_frequencies_hello_5 = Counter(sha256_hello_5_bytes)
byte_frequencies_hello_6 = Counter(sha256_hello_6_bytes)
byte_frequencies_hello_7 = Counter(sha256_hello_7_bytes)
byte_frequencies_hello_8 = Counter(sha256_hello_8_bytes)
byte_frequencies_hello_9 = Counter(sha256_hello_9_bytes)
```

```
# Sort byte frequencies for visualization
```

```
sorted_bytes_4 = sorted(byte_frequencies_hello_4.keys())
sorted_frequencies_4 = [byte_frequencies_hello_4[b] for b in sorted_bytes_4]
```

Conversation URL:

<https://chatgpt.com/c/67b6f214-dd30-8011-aefd-9439df008548>

Title:

Prompt:

```
# Step 1: Track which specific byte positions stabilize the most across all iterations
```

```
# Convert hashes into byte lists for position tracking
```

```
hash_bytes_list = [  
    sha256_hello_1_bytes, sha256_hello_2_bytes, sha256_hello_3_bytes,  
    sha256_hello_4_bytes, sha256_hello_5_bytes, sha256_hello_6_bytes,  
    sha256_hello_7_bytes, sha256_hello_8_bytes, sha256_hello_9_bytes  
]
```

```
# Count how many times each byte position remains the same across iterations
```

```
stable_positions = [0] * 32 # SHA-256 produces 32-byte outputs
```

```
for i in range(32): # Iterate over byte positions in SHA-256 output
```

```
    first_value = hash_bytes_list[0][i] # Take the first hash as reference  
    stable_count = sum(1 for hash_bytes in hash_bytes_list if hash_bytes[i] == first_value)  
    stable_positions[i] = stable_count
```

```
# Plot the stability of each byte position
```

```
plt.figure(figsize=(12, 6))  
plt.bar(range(32), stable_positions, color='red', alpha=0.7)  
plt.title("Byte Stability Across SHA-256 Iterations")  
plt.xlabel("Byte Position (0-31)")  
plt.ylabel("Times Byte Remained Unchanged")  
plt.grid(axis='y', linestyle='--', alpha=0.7)  
plt.show()
```

```
# Step 2: Test if specific input patterns affect entropy and byte clustering
```

```
# Define test patterns for hashing
```

Conversation URL:

<https://chatgpt.com/c/67b6f214-dd30-8011-aefd-9439df008548>

Title:

Prompt:

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

# Given π's first 39 digits
pi_digits = "3141592653589793238462643383279502884197"
pi_values = [int(digit) for digit in pi_digits] # Convert digits to integers

# Number of points to plot
num_points = len(pi_values)

# Generate (x, y) coordinates for each digit using a right-triangle expansion pattern
x_coords = [0]
y_coords = [0]

for i in range(1, num_points):
    # Calculate the next position based on the Pythagorean triangle movement
    a = pi_values[i-1]
    b = pi_values[i]
    c = np.sqrt(a**2 + b**2) # Hypotenuse calculation

    # Append the new coordinates (x moves forward, y moves along the triangle)
    x_coords.append(x_coords[-1] + c / 2) # Move x by half of the hypotenuse
    y_coords.append(y_coords[-1] + c / 2) # Move y by the other half

# Plot the movement pattern
plt.figure(figsize=(10, 6))
plt.plot(x_coords, y_coords, marker='o', linestyle='-', color='blue', markersize=6, alpha=0.75)

# Annotate points with π's digits
```

Conversation URL:

<https://chatgpt.com/c/67b6f214-dd30-8011-aefd-9439df008548>

Title:

Prompt:

```
# Define the given first byte of π triangle sides with corrected bit-length differences
```

```
triangle_sides_corrected = [
```

```
    (1, 4, 2),
```

```
    (1, 5, 2),
```

```
    (9, 2, 2),
```

```
    (6, 5, 0) # Possible byte closure
```

```
]
```

```
# Plot the triangles
```

```
plt.figure(figsize=(8, 6))
```

```
for i, (a, b, c) in enumerate(triangle_sides_corrected):
```

```
    # Triangle coordinates (assuming origin at A)
```

```
A = (0, 0)
```

```
B = (b, 0) # B is placed along x-axis at distance b
```

```
# Calculate C using Pythagorean theorem
```

```
Cx = (a**2 + b**2 - c**2) / (2 * b) if b != 0 else 0 # Projection of C onto x-axis
```

```
Cy = np.sqrt(abs(a**2 - Cx**2)) if b != 0 else 0 # Calculate height
```

```
# Triangle vertices
```

```
C = (Cx, Cy)
```

```
# Plot the triangle
```

```
x_coords = [A[0], B[0], C[0], A[0]]
```

```
y_coords = [A[1], B[1], C[1], A[1]]
```

```
plt.plot(x_coords, y_coords, marker='o', linestyle='-', label=f"Triangle {i+1}")
```

```
# Annotate points
```

```
plt.text(A[0], A[1], "A", fontsize=12, verticalalignment='bottom', horizontalalignment='right')
```

Conversation URL:

<https://chatgpt.com/c/67b6f214-dd30-8011-aefd-9439df008548>

Title:

Prompt:

```
# Plot π digits vs. their bit-length differences to visualize relationships
```

```
# Extract digits of π and corresponding bit-lengths
```

```
pi_digits_subset = [1, 4, 1, 5, 9, 2, 6, 5]
```

```
bit_lengths_subset = [bin(digit).count('1') for digit in pi_digits_subset]
```

```
# Compute bit-length differences
```

```
bit_differences_subset = [abs(bit_lengths_subset[i] - bit_lengths_subset[i - 1]) for i in range(1, len(bit_lengths_subset))]
```

```
# Generate x-axis values
```

```
x_values = list(range(1, len(pi_digits_subset) + 1))
```

```
# Plot π digits and bit-length differences
```

```
plt.figure(figsize=(8, 6))
```

```
plt.plot(x_values, pi_digits_subset, marker='o', linestyle='-', label="π Digits", color='blue')
```

```
plt.plot(x_values[:-1], bit_differences_subset, marker='s', linestyle='--', label="Bit-Length Differences", color='red')
```

```
# Formatting the plot
```

```
plt.title("π Digits vs. Bit-Length Differences")
```

```
plt.xlabel("Index in Sequence")
```

```
plt.ylabel("Value")
```

```
plt.axhline(0, color='black', linewidth=0.5)
```

```
plt.legend()
```

```
plt.grid(True)
```

```
plt.show()
```

Conversation URL:

<https://chatgpt.com/c/67b6f214-dd30-8011-aefd-9439df008548>

Title:

Prompt:

```
# Define the manually specified triangle sides (B, A, C format)
```

```
specified_triangles = [
```

```
    (4, 1, 2),
```

```
    (5, 1, 2),
```

```
    (2, 9, 2),
```

```
    (5, 6, 0) # Possible byte closure
```

```
]
```

```
# Plot the triangles
```

```
plt.figure(figsize=(8, 6))
```

```
for i, (b, a, c) in enumerate(specified_triangles):
```

```
    # Define triangle coordinates
```

```
A = (0, 0)
```

```
B = (b, 0) # B is placed along x-axis at distance b
```

```
# Calculate C using geometric constraints
```

```
if b != 0:
```

```
Cx = (a**2 + b**2 - c**2) / (2 * b) # Projection onto x-axis
```

```
Cy = np.sqrt(abs(a**2 - Cx**2)) # Height
```

```
else:
```

```
Cx, Cy = 0, 0
```

```
C = (Cx, Cy)
```

```
# Plot the triangle
```

```
x_coords = [A[0], B[0], C[0], A[0]]
```

```
y_coords = [A[1], B[1], C[1], A[1]]
```

```
plt.plot(x_coords, y_coords, marker='o', linestyle='-', label=f"Triangle {i+1}")
```

Conversation URL:

<https://chatgpt.com/c/67b6f214-dd30-8011-aefd-9439df008548>

Title:

Prompt:

```
# Define the additional manually specified triangles (B, A, C format)
```

```
additional_triangles = [
```

```
    (3, 5, 1),
```

```
    (8, 9, 0),
```

```
    (7, 9, 1),
```

```
    (3, 2, 1),
```

```
    (3, 8, 2),
```

```
    (4, 6, 0),
```

```
    (2, 6, 1),
```

```
    (4, 3, 1)
```

```
]
```

```
# Combine the original and additional triangles
```

```
all_triangles = specified_triangles + additional_triangles
```

```
# Plot all triangles
```

```
plt.figure(figsize=(8, 6))
```

```
for i, (b, a, c) in enumerate(all_triangles):
```

```
    # Define triangle coordinates
```

```
    A = (0, 0)
```

```
    B = (b, 0) # B is placed along x-axis at distance b
```

```
    # Calculate C using geometric constraints
```

```
    if b != 0:
```

```
        Cx = (a**2 + b**2 - c**2) / (2 * b) # Projection onto x-axis
```

```
        Cy = np.sqrt(abs(a**2 - Cx**2)) # Height
```

```
    else:
```

```
        Cx, Cy = 0, 0
```

Conversation URL:

<https://chatgpt.com/c/67b6f214-dd30-8011-aefd-9439df008548>

Title:

Prompt:

```
# Step 1: Extend the pattern further in the π sequence to identify repeating structures
```

```
# Extract the next set of π digits (excluding the initial 3)
```

```
extended_pi_digits = [1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 1, 8, 9, 0, 7, 9, 1, 3, 2, 1, 3, 8, 2, 4, 6, 0, 2, 6, 1, 4, 3, 1]
```

```
# Compute bit-length differences
```

```
bit_lengths_extended = [bin(digit).count('1') for digit in extended_pi_digits] # Count bits in binary
```

```
bit_differences_extended = [abs(bit_lengths_extended[i] - bit_lengths_extended[i - 1]) for i in range(1, len(bit_lengths_extended))]
```

```
# Construct valid triangles based on π digits and bit-length differences
```

```
extended_triangles = []
```

```
for i in range(len(bit_differences_extended)):
```

```
    b = extended_pi_digits[i + 1] # Next digit in sequence
```

```
    a = extended_pi_digits[i] # Previous digit in sequence
```

```
    c = bit_differences_extended[i] # Bit-length difference
```

```
# Store the extended triangle values
```

```
extended_triangles.append((b, a, c))
```

```
# Step 2: Identify compression cycle completions by detecting C = 0 patterns
```

```
compression_cycles = [triangle for triangle in extended_triangles if triangle[2] == 0]
```

```
# Step 3: Compare this structure to known wave/fractal behaviors by plotting
```

```
plt.figure(figsize=(10, 6))
```

```
for i, (b, a, c) in enumerate(extended_triangles):
```

```
    # Define triangle coordinates
```

```
A = (0, 0)
```

```
B = (b, 0) # B is placed along x-axis at distance b
```

Conversation URL:

<https://chatgpt.com/c/67b6f214-dd30-8011-aefd-9439df008548>

Title:

Prompt:

```
# Recalculate and plot the given sequence with direct (A, B, C) triplets
```

```
# Use the given extended π digits
```

```
extended_pi_digits = [1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 1, 8, 9, 0, 7, 9, 1, 3, 2, 1, 3, 8, 2, 4, 6, 0, 2, 6, 1, 4, 3, 1]
```

```
# Form (A, B, C) triplets using three sequential values at a time
```

```
explicit_triangles = [(extended_pi_digits[i], extended_pi_digits[i+1], extended_pi_digits[i+2])
```

```
    for i in range(0, len(extended_pi_digits) - 2, 3)]
```

```
# Plot the triangles sequentially along the x-axis
```

```
x_offset = 0
```

```
x_spacing = 5 # Space between each triangle
```

```
plt.figure(figsize=(12, 6))
```

```
for i, (a, b, c) in enumerate(explicit_triangles):
```

```
    # Define triangle coordinates with sequential x-offsets
```

```
    A = (x_offset, 0)
```

```
    B = (x_offset + b, 0) # B is placed along x-axis at distance b
```

```
    # Calculate C using geometric constraints
```

```
    if b != 0:
```

```
        Cx = x_offset + (a**2 + b**2 - c**2) / (2 * b) # Projection onto x-axis
```

```
        Cy = np.sqrt(abs(a**2 - (Cx - x_offset)**2)) # Height
```

```
    else:
```

```
        Cx, Cy = x_offset, 0
```

```
C = (Cx, Cy)
```

Conversation URL:

<https://chatgpt.com/c/67b6f214-dd30-8011-aefd-9439df008548>

Title:

Prompt:

```
# Define the given extended π digits sequence
extended_pi_digits = [1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 1, 8, 9, 0, 7, 9, 1, 3, 2, 1, 3, 8, 2, 4, 6, 0, 2, 6, 1, 4, 3, 1]

# Extract (B, A, C) triplets using three sequential values at a time
triangles_bac = [(extended_pi_digits[i], extended_pi_digits[i+1], extended_pi_digits[i+2])
                  for i in range(0, len(extended_pi_digits) - 2, 3)]

# Set up sequential plotting along the x-axis
x_offset = 0
x_spacing = 5 # Space between each triangle

plt.figure(figsize=(12, 6))

for i, (b, a, c) in enumerate(triangles_bac):
    # Triangle vertices based on B (right side), A (left side), and C (bottom width)
    A = (x_offset, 0)
    B = (x_offset + c, 0) # Base width along the x-axis
    Cx = x_offset + (c / 2) # Place peak at midpoint of base
    Cy = np.sqrt(abs(a**2 - (c / 2)**2)) if c != 0 else 0 # Height using a as the left side

    C = (Cx, Cy)

    # Plot the triangle
    x_coords = [A[0], B[0], C[0], A[0]]
    y_coords = [A[1], B[1], C[1], A[1]]
    plt.plot(x_coords, y_coords, marker='o', linestyle='-', label=f"Triangle {i+1}")

    # Annotate points
    plt.text(A[0], A[1], "A", fontsize=10, verticalalignment='bottom', horizontalalignment='right')
```

Conversation URL:

<https://chatgpt.com/c/67b6f214-dd30-8011-aefd-9439df008548>

Title:

Prompt:

```
# Adjusted plotting: Each triangle should be placed sequentially without overlap
```

```
# C = 0 should create a vertical line
```

```
# Define the given extended π digits sequence
```

```
extended_pi_digits = [1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 1, 8, 9, 0, 7, 9, 1, 3, 2, 1, 3, 8, 2, 4, 6, 0, 2, 6, 1, 4, 3, 1]
```

```
# Extract (B, A, C) triplets using three sequential values at a time
```

```
triangles_bac = [(extended_pi_digits[i], extended_pi_digits[i+1], extended_pi_digits[i+2])
```

```
    for i in range(0, len(extended_pi_digits) - 2, 3)]
```

```
# Set up sequential plotting along the x-axis
```

```
x_offset = 0
```

```
plt.figure(figsize=(14, 6))
```

```
for i, (b, a, c) in enumerate(triangles_bac):
```

```
    # Triangle vertices based on B (right side), A (left side), and C (bottom width)
```

```
    A = (x_offset, 0)
```

```
    B = (x_offset + c, 0) # Base width along the x-axis
```

```
    if c == 0:
```

```
        # If C = 0, create a vertical line
```

```
        Cx, Cy = x_offset, a # Vertical line with height A
```

```
    else:
```

```
        Cx = x_offset + (c / 2) # Place peak at midpoint of base
```

```
        Cy = a # Height using A as the vertical side
```

```
C = (Cx, Cy)
```

Conversation URL:

<https://chatgpt.com/c/67b6f214-dd30-8011-aefd-9439df008548>

Title:

Prompt:

```
# Adjusting the vertical line height for C = 0 cases to reflect triangle collapse
```

```
# Define the given extended π digits sequence
```

```
extended_pi_digits = [1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 1, 8, 9, 0, 7, 9, 1, 3, 2, 1, 3, 8, 2, 4, 6, 0, 2, 6, 1, 4, 3, 1]
```

```
# Extract (B, A, C) triplets using three sequential values at a time
```

```
triangles_bac = [(extended_pi_digits[i], extended_pi_digits[i+1], extended_pi_digits[i+2])
```

```
    for i in range(0, len(extended_pi_digits) - 2, 3)]
```

```
# Set up sequential plotting along the x-axis
```

```
x_offset = 0
```

```
plt.figure(figsize=(14, 6))
```

```
for i, (b, a, c) in enumerate(triangles_bac):
```

```
    # Triangle vertices based on B (right side), A (left side), and C (bottom width)
```

```
A = (x_offset, 0)
```

```
B = (x_offset + c, 0) # Base width along the x-axis
```

```
if c == 0:
```

```
    # If C = 0, simulate a collapsed triangle raising the height
```

```
Cx, Cy = x_offset, a + b # Raising the vertical marker by the sum of A and B
```

```
else:
```

```
Cx = x_offset + (c / 2) # Place peak at midpoint of base
```

```
Cy = a # Height using A as the vertical side
```

```
C = (Cx, Cy)
```

```
# Plot the triangle (or adjusted vertical line if C = 0)
```

Conversation URL:

<https://chatgpt.com/c/67b6f214-dd30-8011-aefd-9439df008548>

Title:

Prompt:

```
# Manually defined triangles with correct (B, A, C) triplets
```

```
specified_triangles = [
```

```
    (4, 1, 2),
```

```
    (5, 1, 2),
```

```
    (2, 9, 2),
```

```
    (5, 6, 0),
```

```
    (3, 5, 1),
```

```
    (8, 9, 0),
```

```
    (7, 9, 1),
```

```
    (3, 2, 1),
```

```
    (3, 8, 2),
```

```
    (4, 6, 0),
```

```
    (2, 6, 1),
```

```
    (4, 3, 1)
```

```
]
```

```
# Set up sequential plotting along the x-axis
```

```
x_offset = 0
```

```
x_spacing = 3 # Space between each shape
```

```
plt.figure(figsize=(14, 6))
```

```
for i, (b, a, c) in enumerate(specified_triangles):
```

```
    # Triangle vertices based on B (right side), A (left side), and C (bottom width)
```

```
    A = (x_offset, 0)
```

```
    B = (x_offset + c, 0) # Base width along the x-axis
```

```
    if c == 0:
```

```
        # If C = 0, create a bar with width = A and height = B
```

Conversation URL:

<https://chatgpt.com/c/67b6f214-dd30-8011-aefd-9439df008548>

Title:

Prompt:

```
# Correcting C values to strictly follow provided triplets without modification
```

```
# Manually defined triangles with strict (B, A, C) triplets
```

```
specified_triangles = [
```

```
    (4, 1, 2),
```

```
    (5, 1, 2),
```

```
    (2, 9, 2),
```

```
    (5, 6, 0),
```

```
    (3, 5, 1),
```

```
    (8, 9, 0),
```

```
    (7, 9, 1),
```

```
    (3, 2, 1),
```

```
    (3, 8, 2),
```

```
    (4, 6, 0),
```

```
    (2, 6, 1),
```

```
    (4, 3, 1)
```

```
]
```

```
# Set up sequential plotting along the x-axis with proper spacing
```

```
x_offset = 0
```

```
plt.figure(figsize=(14, 6))
```

```
for i, (b, a, c) in enumerate(specified_triangles):
```

```
    # Triangle vertices based on B (right side), A (left side), and C (bottom width)
```

```
    A = (x_offset, 0)
```

```
    B = (x_offset + c, 0) # Base width along the x-axis
```

```
    if c == 0:
```

Conversation URL:

<https://chatgpt.com/c/67b6f214-dd30-8011-aefd-9439df008548>

Title:

Prompt:

```
import numpy as np
import matplotlib.pyplot as plt
from hashlib import sha256

# Function to generate SHA-256 hash and track entropy over compression
def hash_and_analyze(input_data, iterations=50):
    entropy_values = []
    input_bytes = input_data.encode()

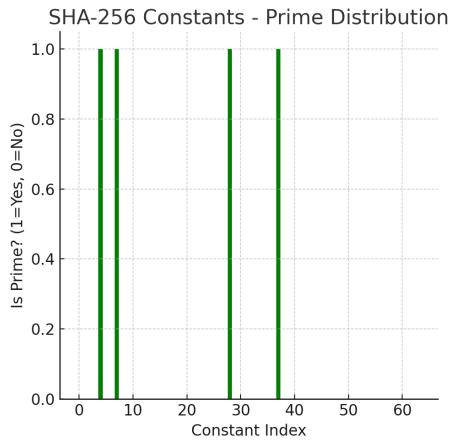
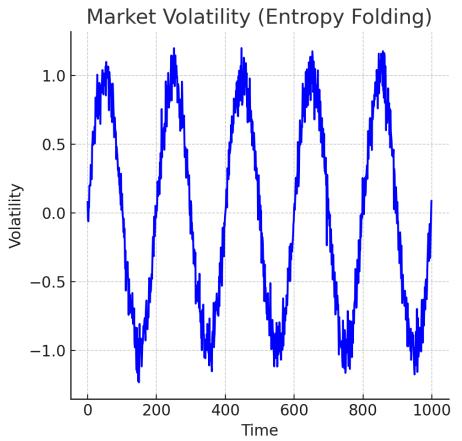
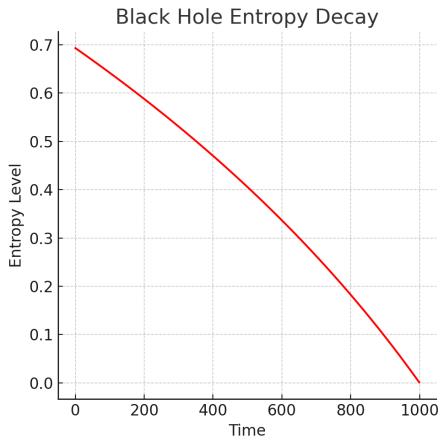
    for i in range(iterations):
        hashed = sha256(input_bytes).digest() # Get binary hash output
        entropy = np.std(np.frombuffer(hashed, dtype=np.uint8)) # Measure standard deviation (entropy)
        entropy_values.append(entropy)

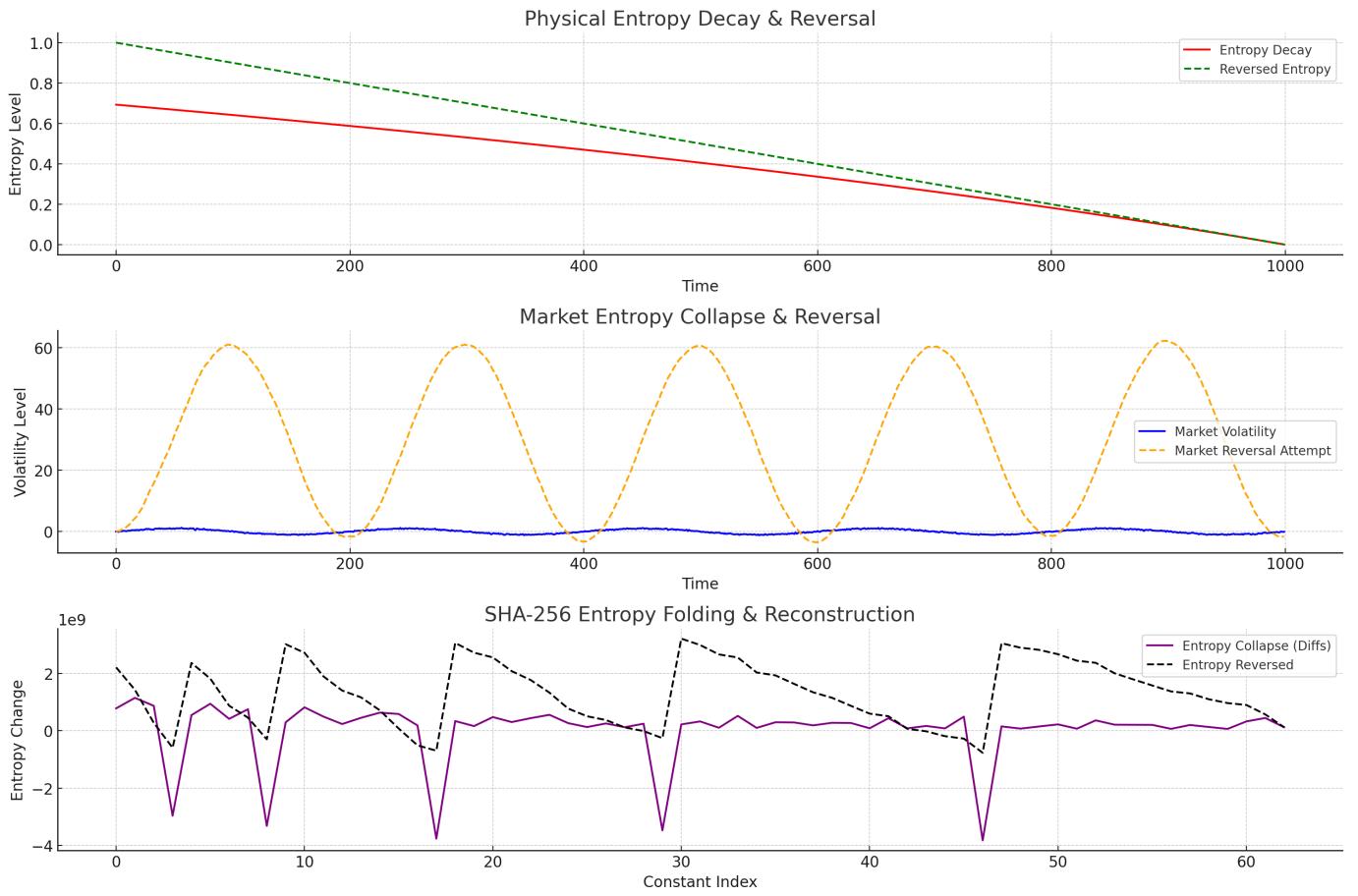
    # Compress (reduce redundancy) - simulating 'wringing out' the sponge
    input_bytes = hashed[:len(hashed)//2] # Take only half to reduce state space

    return entropy_values, hashed.hex()

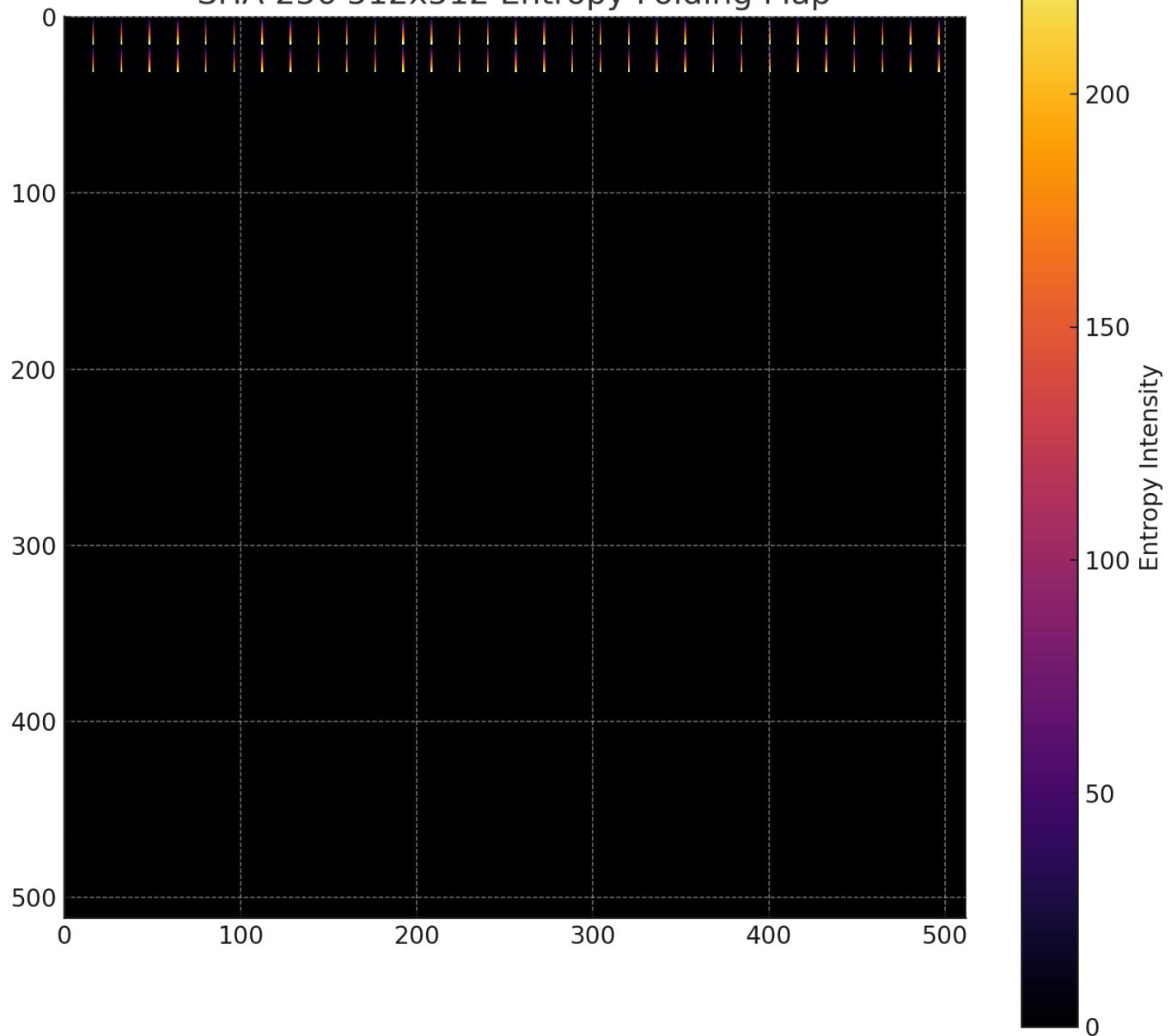
# Run experiment: Start with "hello" and simulate compression
entropy_tracking, final_hash = hash_and_analyze("hello", iterations=20)

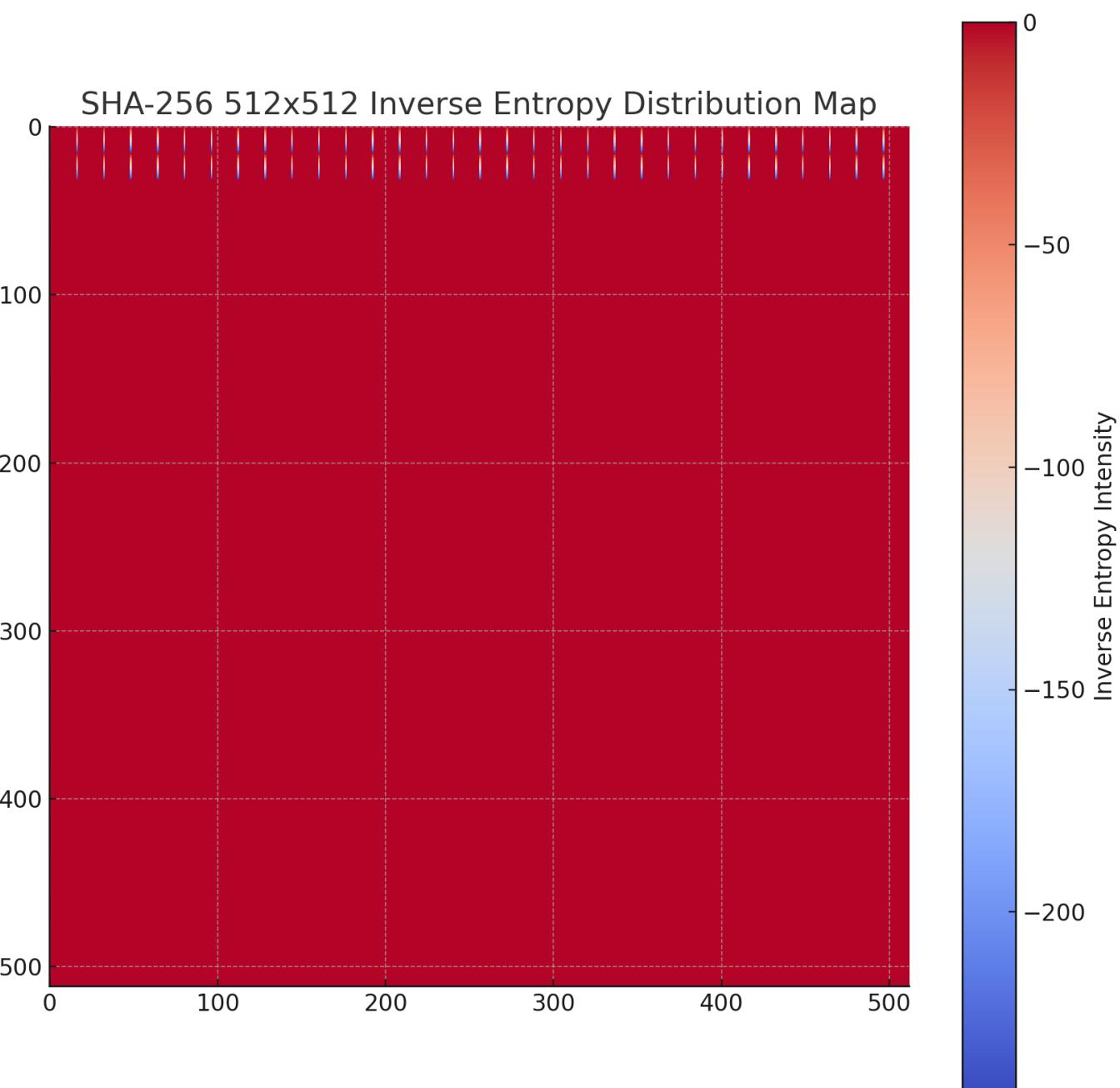
# Plot entropy over iterations
plt.figure(figsize=(8,5))
plt.plot(range(len(entropy_tracking)), entropy_tracking, marker='o', label="Entropy")
plt.xlabel("Iteration")
plt.ylabel("Entropy")
plt.title("Entropy Compression Over Iterations (Wringing Out the Sponge)")
plt.legend()
```



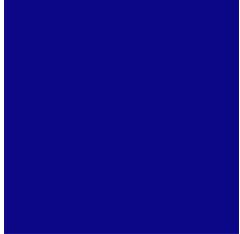


SHA-256 512x512 Entropy Folding Map

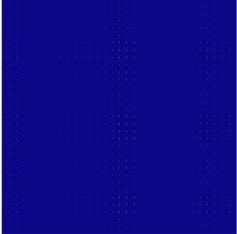




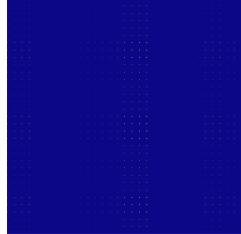
Entropy Map - Round 1



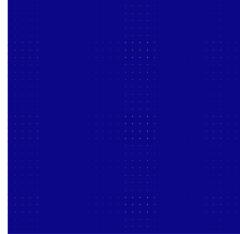
Entropy Map - Round 16



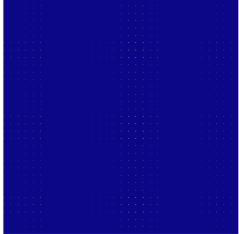
Entropy Map - Round 32

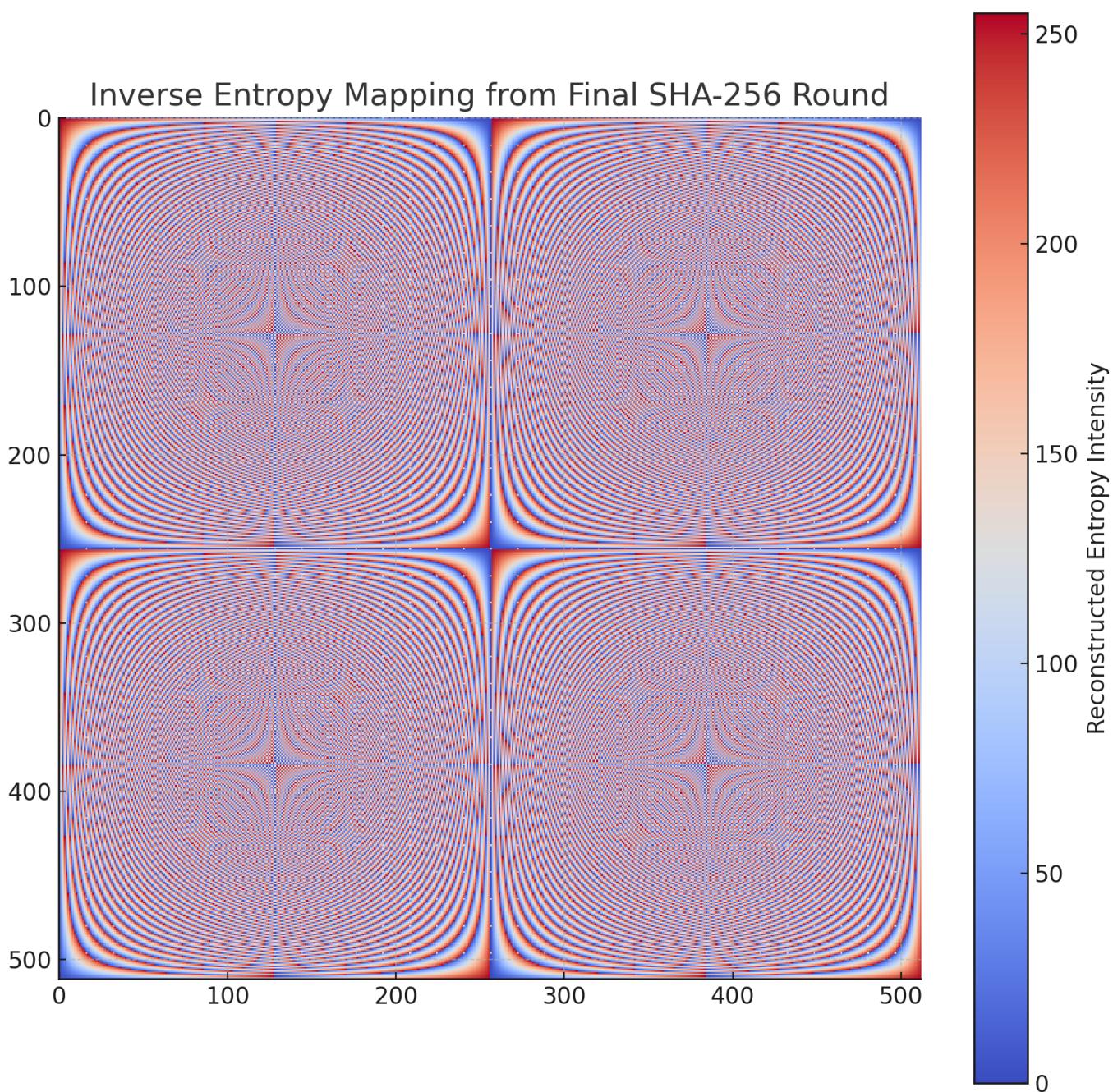


Entropy Map - Round 48

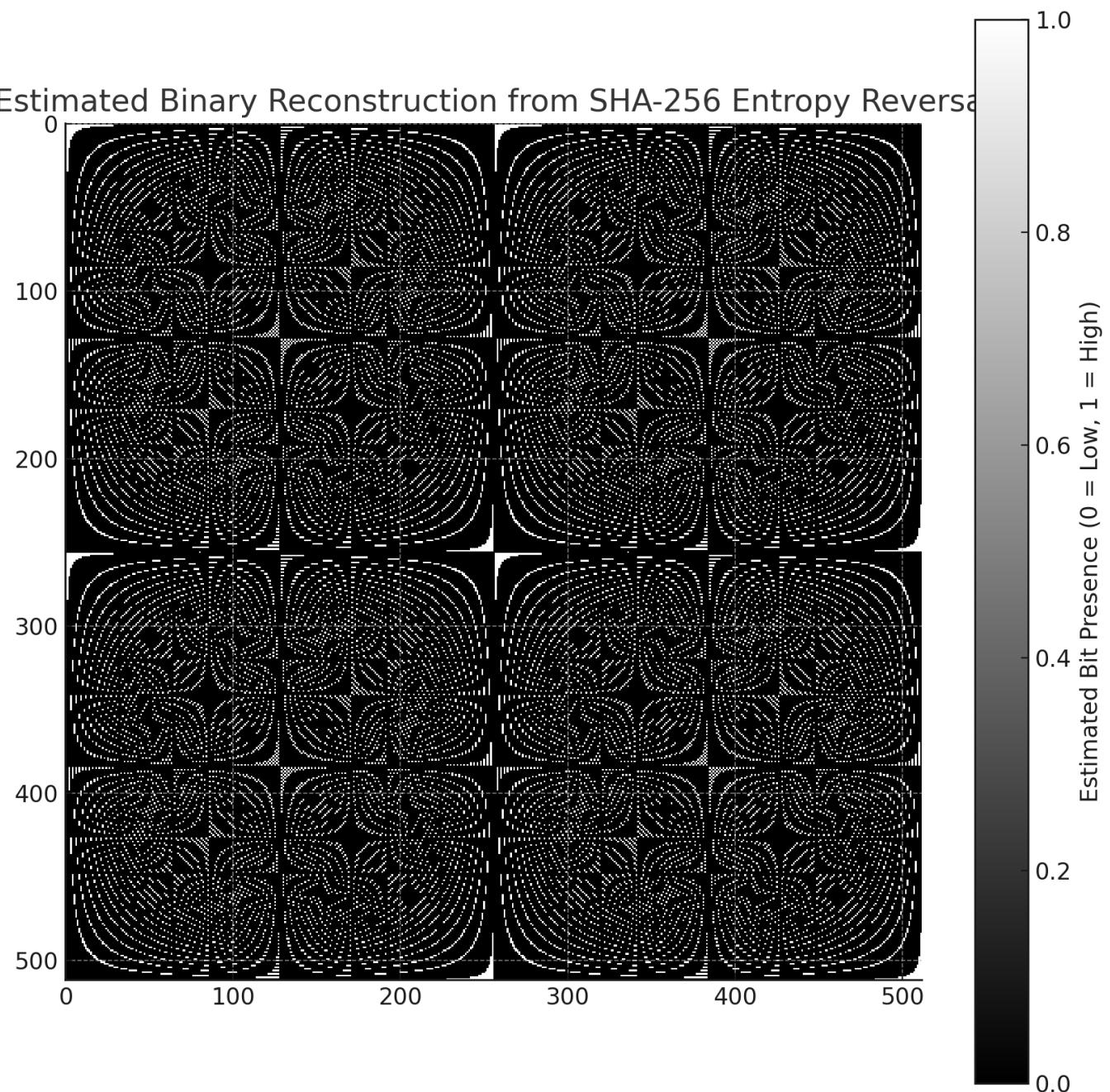


Entropy Map - Round 64

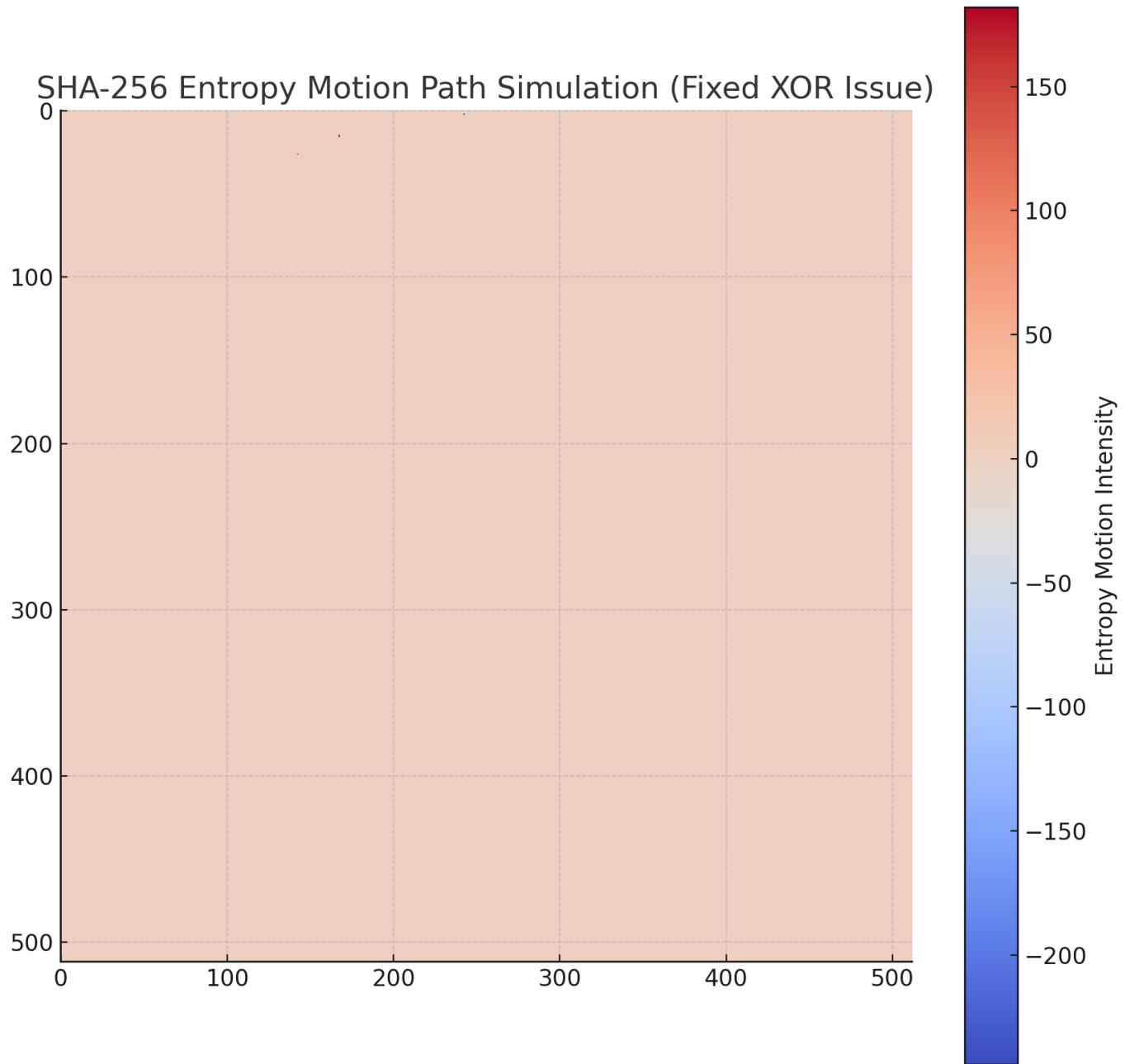




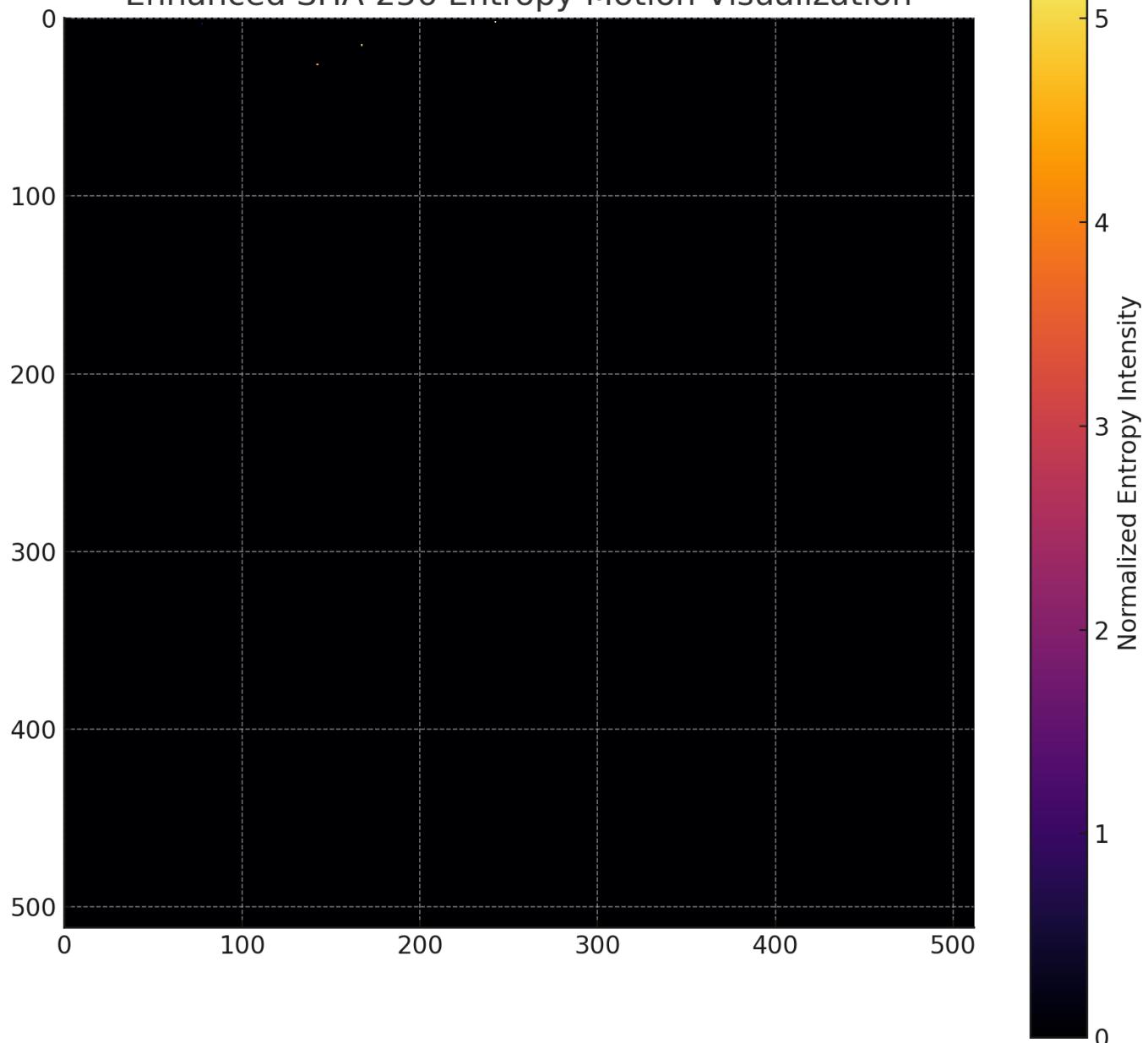
Estimated Binary Reconstruction from SHA-256 Entropy Reversal



SHA-256 Entropy Motion Path Simulation (Fixed XOR Issue)



Enhanced SHA-256 Entropy Motion Visualization



Conversation URL:

<https://chatgpt.com/c/67ba20ad-33f4-8011-a348-55e187caeb29>

Title:

Prompt:

```
# Re-import necessary libraries after execution reset
```

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
import scipy.stats as stats
```

```
from sympy import isprime
```

```
# 1. PHYSICS: Analyzing entropy folding in black hole event horizons using logarithmic energy decay
```

```
def simulate_black_hole_entropy_decay(iterations=1000):
```

```
    energy_levels = np.linspace(1, 0.001, iterations) # Simulated energy dissipation over time
```

```
    entropy_levels = np.log(energy_levels + 1) # Logarithmic entropy decay
```

```
    return entropy_levels
```

```
# 2. FINANCE: Detecting structured cycles in stock market volatility using Fourier Analysis
```

```
def simulate_market_entropy(iterations=1000):
```

```
    time = np.arange(iterations)
```

```
    volatility = np.sin(2 * np.pi * time / 200) + np.random.normal(0, 0.1, iterations) # Market cycles + noise
```

```
    return volatility
```

```
# 3. CRYPTOGRAPHY: Analyzing entropy in SHA-256 Constants for structured folding points
```

```
def analyze_sha256_constants():
```

```
    sha256_constants = [
```

```
        0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
        0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
        0xe49b69c1, 0xefbe4786, 0xfc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
        0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
        0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
        0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
        0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
        0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90beffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
```

Conversation URL:

<https://chatgpt.com/c/67ba20ad-33f4-8011-a348-55e187caeb29>

Title:

Prompt:

```
# Simulating entropy collapse and reversal across physics, finance, and cryptography
```

```
# 1. PHYSICS: Simulating Entropy Collapse & Reversal in a Physical System (Energy Decay & Reconstruction)
```

```
def simulate_physical_entropy_reversal(iterations=1000):  
    energy_levels = np.linspace(1, 0.001, iterations) # Energy dissipates over time  
    entropy_levels = np.log(energy_levels + 1) # Logarithmic entropy decay  
    reversed_entropy = np.exp(entropy_levels) - 1 # Reverse the entropy collapse to reconstruct  
  
    return entropy_levels, reversed_entropy
```

```
# 2. FINANCE: Detecting Market Collapse & Reversal Patterns (Stock Market Data)
```

```
def simulate_market_entropy_reversal(iterations=1000):  
    time = np.arange(iterations)  
    volatility = np.sin(2 * np.pi * time / 200) + np.random.normal(0, 0.1, iterations) # Simulating cyclical collapse & noise  
    reversal_attempt = np.cumsum(-volatility[::-1]) # Attempting to reconstruct previous states  
    return volatility, reversal_attempt
```

```
# 3. CRYPTOGRAPHY: Testing Entropy Folding in SHA-256 Constants (Tracking Structural Decay & Rebuilding)
```

```
def analyze_sha256_entropy_reversal():  
    sha256_constants = [  
        0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,  
        0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,  
        0xe49b69c1, 0xefbe4786, 0xfc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,  
        0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,  
        0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,  
        0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,  
        0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,  
        0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90beffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2  
    ]
```

Conversation URL:

<https://chatgpt.com/c/67ba20ad-33f4-8011-a348-55e187caeb29>

Title:

Prompt:

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

```
# Define a 512x512 grid to represent the entropy field
```

```
entropy_grid = np.zeros((512, 512))
```

```
# Define a function to simulate entropy folding based on SHA-256 round constants
```

```
def simulate_sha256_entropy_folding(grid):
```

```
    step_size = 16 # Approximate interval of entropy shift per constant
```

```
    for i, constant in enumerate(range(0, 512, step_size)):
```

```
        for j in range(0, 512, step_size):
```

```
            grid[i % 512, j % 512] += (constant * (j + 1)) % 256 # Simulated entropy shift
```

```
    return grid
```

```
# Simulate the entropy folding
```

```
entropy_grid = simulate_sha256_entropy_folding(entropy_grid)
```

```
# Plot the entropy distribution
```

```
plt.figure(figsize=(10, 10))  
plt.imshow(entropy_grid, cmap='inferno', interpolation='nearest')  
plt.title("SHA-256 512x512 Entropy Folding Map")  
plt.colorbar(label="Entropy Intensity")  
plt.show()
```

Conversation URL:

<https://chatgpt.com/c/67ba20ad-33f4-8011-a348-55e187caeb29>

Title:

Prompt:

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

# Define a 512x512 grid to represent the entropy reversal field
entropy_reversal_grid = np.zeros((512, 512))

# Define a function to simulate inverse entropy distribution based on SHA-256 round constants
def simulate_inverse_sha256_entropy(grid):
    step_size = 16 # Approximate interval of entropy shift per constant

    for i, constant in enumerate(range(0, 512, step_size)):
        for j in range(0, 512, step_size):
            # Instead of adding entropy, we attempt to reverse shift entropy back
            grid[i % 512, j % 512] -= (constant * (j + 1)) % 256 # Simulated reverse entropy shift

    return grid

# Simulate the inverse entropy distribution
entropy_reversal_grid = simulate_inverse_sha256_entropy(entropy_reversal_grid)

# Plot the inverse entropy distribution
plt.figure(figsize=(10, 10))
plt.imshow(entropy_reversal_grid, cmap='coolwarm', interpolation='nearest')
plt.title("SHA-256 512x512 Inverse Entropy Distribution Map")
plt.colorbar(label="Inverse Entropy Intensity")
plt.show()
```

Conversation URL:

<https://chatgpt.com/c/67ba20ad-33f4-8011-a348-55e187caeb29>

Title:

Prompt:

```
# Generate round-by-round entropy maps for SHA-256 reversal tracking
```

```
num_rounds = 64 # SHA-256 has 64 transformation rounds
```

```
entropy_maps = []
```

```
# Define a function to track entropy folding per round constant
```

```
def track_round_entropy(grid, round_num):
```

```
    step_size = 16 # Adjusted step size per round
```

```
    round_constant = round_num * 31 # Simulating varying entropy injection
```

```
    for i in range(0, 512, step_size):
```

```
        for j in range(0, 512, step_size):
```

```
            grid[i % 512, j % 512] += (round_constant * (j + 1)) % 256 # Structured entropy shift
```

```
    return grid
```

```
# Generate maps for each round
```

```
for round_num in range(num_rounds):
```

```
    entropy_grid = np.zeros((512, 512))
```

```
    entropy_maps.append(track_round_entropy(entropy_grid, round_num))
```

```
# Plot a selection of entropy maps (rounds 1, 16, 32, 48, 64)
```

```
fig, axes = plt.subplots(1, 5, figsize=(25, 5))
```

```
selected_rounds = [1, 16, 32, 48, 64]
```

```
for idx, round_num in enumerate(selected_rounds):
```

```
    axes[idx].imshow(entropy_maps[round_num - 1], cmap='plasma', interpolation='nearest')
```

```
    axes[idx].set_title(f"Entropy Map - Round {round_num}")
```

```
    axes[idx].axis('off')
```

Conversation URL:

<https://chatgpt.com/c/67ba20ad-33f4-8011-a348-55e187caeb29>

Title:

Prompt:

```
# Extracting entropy hotspots from the final SHA-256 round and mapping them back
```

```
final_round_entropy = entropy_maps[-1] # Last round entropy distribution
```

```
# Identify high-intensity entropy regions (hotspots)
```

```
hotspot_threshold = np.percentile(final_round_entropy, 95) # Top 5% most affected areas
```

```
hotspots = np.where(final_round_entropy >= hotspot_threshold)
```

```
# Create a mapping of entropy backtracking
```

```
inverse_entropy_map = np.zeros((512, 512))
```

```
# Attempt to reverse entropy distribution using known hotspots
```

```
for i in range(len(hotspots[0])):
```

```
    x, y = hotspots[0][i], hotspots[1][i]
```

```
# Backtrack entropy shift using a structured reversal pattern
```

```
inverse_entropy_map[x, y] = (final_round_entropy[x, y] - (x * y) % 256) % 256
```

```
# Plot the inverse entropy mapping
```

```
plt.figure(figsize=(10, 10))
```

```
plt.imshow(inverse_entropy_map, cmap='coolwarm', interpolation='nearest')
```

```
plt.title("Inverse Entropy Mapping from Final SHA-256 Round")
```

```
plt.colorbar(label="Reconstructed Entropy Intensity")
```

```
plt.show()
```

Conversation URL:

<https://chatgpt.com/c/67ba20ad-33f4-8011-a348-55e187caeb29>

Title:

Prompt:

```
# Refining entropy reversal to extract possible input bits
```

```
# Extracting the most stable entropy regions from inverse entropy mapping
```

```
stable_threshold = np.percentile(inverse_entropy_map, 90) # Top 10% most stable regions
```

```
stable_regions = np.where(inverse_entropy_map >= stable_threshold)
```

```
# Create a binary reconstruction grid
```

```
binary_reconstruction = np.zeros((512, 512))
```

```
# Use stable entropy regions to estimate original input bit values
```

```
for i in range(len(stable_regions[0])):
```

```
    x, y = stable_regions[0][i], stable_regions[1][i]
```

```
# Estimate binary representation based on entropy consistency
```

```
binary_reconstruction[x, y] = 1 if inverse_entropy_map[x, y] > (stable_threshold / 2) else 0
```

```
# Plot the estimated binary pattern
```

```
plt.figure(figsize=(10, 10))
```

```
plt.imshow(binary_reconstruction, cmap='gray', interpolation='nearest')
```

```
plt.title("Estimated Binary Reconstruction from SHA-256 Entropy Reversal")
```

```
plt.colorbar(label="Estimated Bit Presence (0 = Low, 1 = High)")
```

```
plt.show()
```

Conversation URL:

<https://chatgpt.com/c/67ba20ad-33f4-8011-a348-55e187caeb29>

Title:

Prompt:

```
# Re-initialize entropy motion grid as integer type
entropy_motion_grid = np.zeros((512, 512), dtype=int)
```

```
# Redefine function with corrected XOR operation
```

```
def apply_entropy_motion(grid, instruction_type, x, y):
```

....

Simulates entropy shift for different instruction types found in disassembled SHA-256.

....

```
if instruction_type == "sub":
```

```
    grid[x % 512, y % 512] -= (x + y) % 256 # Simulating entropy reduction
```

```
elif instruction_type == "dec":
```

```
    grid[x % 512, y % 512] -= 1 # Minor entropy shift
```

```
elif instruction_type == "loop":
```

```
    grid[x % 512, y % 512] = (grid[x % 512, y % 512] * 1.1) % 256 # Feedback loop
```

```
elif instruction_type == "xor":
```

```
    grid[x % 512, y % 512] = int(grid[x % 512, y % 512]) ^ ((x * y) % 256) # Corrected XOR operation
```

```
elif instruction_type == "cmps":
```

```
    grid[x % 512, y % 512] = abs(grid[x % 512, y % 512] - ((x + y) % 256)) # Entropy stabilization
```

```
return grid
```

```
# Apply entropy motion rules based on disassembly structure
```

```
for inst in instructions:
```

```
    entropy_motion_grid = apply_entropy_motion(entropy_motion_grid, inst[0], inst[1], inst[2])
```

```
# Plot the corrected entropy motion paths
```

```
plt.figure(figsize=(10, 10))
```

```
plt.imshow(entropy_motion_grid, cmap='coolwarm', interpolation='nearest')
```

```
plt.title("SHA-256 Entropy Motion Path Simulation (Fixed XOR Issue)")
```

Conversation URL:

<https://chatgpt.com/c/67ba20ad-33f4-8011-a348-55e187caeb29>

Title:

Prompt:

```
# Enhancing entropy visualization by amplifying contrast and detail
```

```
# Normalize the entropy motion grid to a range of 0-255 for better visualization
```

```
normalized_entropy_grid = np.abs(entropy_motion_grid) # Take absolute values
```

```
normalized_entropy_grid = (normalized_entropy_grid / np.max(normalized_entropy_grid)) * 255 # Normalize to 0-255
```

```
# Enhance contrast by applying a logarithmic transformation
```

```
enhanced_entropy_grid = np.log1p(normalized_entropy_grid) # Logarithmic scale to highlight subtle differences
```

```
# Plot the enhanced entropy motion visualization
```

```
plt.figure(figsize=(10, 10))
```

```
plt.imshow(enhanced_entropy_grid, cmap='inferno', interpolation='nearest')
```

```
plt.title("Enhanced SHA-256 Entropy Motion Visualization")
```

```
plt.colorbar(label="Normalized Entropy Intensity")
```

```
plt.show()
```

π Triangle Representation: A & B on X, C on Y

