Appendix C: Visualization Library

C.1 Field Visualization Techniques

C.1.1 Intent Field Representations

Vector Field Visualization

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def visualize_intent_field(field_data, time_step):
    """
    Visualize 3D intent field with vector arrows

    Parameters:
    - field_data: 4D array (x, y, z, components)
    - time_step: specific time point to visualize
    """
    fig = plt.figure(figsize=(12, 10))
    ax = fig.add_subplot(111, projection='3d')

    # Create mesh grid
    x, y, z = np.meshgrid(
        np.linspace(0, 10, 20),
        np.linspace(0, 10, 20),
        np.linspace(0, 10, 10)
    )

    # Extract field components
    u = field_data[:, :, :, 0, time_step]
    v = field_data[:, :, :, 1, time_step]
    w = field_data[:, :, :, 2, time_step]

    # Plot vector field
    ax.quiver(x, y, z, u, v, w,
              arrow_length_ratio=0.1,
              colors=plt.cm.viridis(np.linalg.norm([u, v, w], axis=0)))

    ax.set_xlabel('X (Information)')
    ax.set_ylabel('Y (Structure)')
    ax.set_zlabel('Z (Coherence)')
    ax.set_title(f'Intent Field Visualization - Timestep {time_step}')

    return fig
```

Streamline Visualization

python

```python
def visualize_field_streamlines(field_data, time_step):
    """
    Create streamline visualization of intent field flow
    """
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 7))

    # 2D slice at z = center
    z_center = field_data.shape[2] // 2
    slice_data = field_data[:, :, z_center, :, time_step]

    x = np.arange(slice_data.shape[0])
    y = np.arange(slice_data.shape[1])
    X, Y = np.meshgrid(x, y)

    # Streamlines
    ax1.streamplot(X, Y, slice_data[:, :, 0], slice_data[:, :, 1],
                   color=np.sqrt(slice_data[:, :, 0]**2 + slice_data[:, :, 1]**2),
                   cmap='plasma')
    ax1.set_title('Intent Field Streamlines')

    # Field magnitude heatmap
    magnitude = np.sqrt(np.sum(slice_data**2, axis=2))
    im = ax2.imshow(magnitude, cmap='hot', origin='lower')
    ax2.set_title('Field Magnitude')
    plt.colorbar(im, ax=ax2)

    return fig
```

C.1.2 Harmonic Bloom Visualization

Time Evolution of Blooms

python

```python
def visualize_bloom_cascade(complexity_data, bloom_events):
    """
    Visualize the harmonic bloom cascade over time
    """
    fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(15, 15))

    # Main complexity evolution
    time_steps = np.arange(len(complexity_data))
    ax1.plot(time_steps, complexity_data, 'b-', linewidth=2)

    # Mark bloom events
    for bloom_time in bloom_events:
        ax1.axvline(x=bloom_time, color='r', linestyle='--', alpha=0.7)
        ax1.scatter(bloom_time, complexity_data[bloom_time],
                    color='red', s=100, zorder=10)

    ax1.set_xlabel('Time Steps')
    ax1.set_ylabel('Complexity')
    ax1.set_title('Harmonic Bloom Cascade Evolution')
    ax1.grid(True, alpha=0.3)

    # Derivative analysis
    complexity_derivative = np.gradient(complexity_data)
    ax2.plot(time_steps, complexity_derivative, 'g-', linewidth=1.5)
    ax2.axhline(y=0, color='k', linestyle='-', alpha=0.5)
    ax2.set_ylabel('dComplexity/dt')
    ax2.grid(True, alpha=0.3)

    # Phase portrait (complexity vs derivative)
    ax3.plot(complexity_data, complexity_derivative, 'purple', alpha=0.7)
    ax3.scatter(complexity_data[bloom_events],
                complexity_derivative[bloom_events],
                color='red', s=100, zorder=10)
    ax3.set_xlabel('Complexity')
    ax3.set_ylabel('dComplexity/dt')
    ax3.set_title('Phase Portrait')
    ax3.grid(True, alpha=0.3)

    plt.tight_layout()
    return fig
```

3D Bloom Visualization

```python
def visualize_3d_bloom_dynamics(field_data, bloom_events):
    """
    3D visualization of field dynamics during bloom events
    """
    fig = plt.figure(figsize=(15, 12))

    for i, (bloom_idx, bloom_time) in enumerate(zip(bloom_events,
                                                    ['Bloom 1', 'Bloom 2', 'Bloom 3', '
        ax = fig.add_subplot(2, 3, i+1, projection='3d')

        # Extract data around bloom event
        start_t = max(0, bloom_idx - 10)
        end_t = min(len(field_data[0, 0, 0, :]), bloom_idx + 10)

        # Create surface
        x, y = np.meshgrid(range(field_data.shape[0]), range(field_data.shape[1]))

        # Field magnitude at bloom center
        z_center = field_data.shape[2] // 2
        z_data = field_data[:, :, z_center, 3, bloom_idx]  # Coherence component

        surf = ax.plot_surface(x, y, z_data, cmap='viridis', alpha=0.8)

        ax.set_title(f'{bloom_time} - Timestep {bloom_idx}')
        ax.set_xlabel('X')
        ax.set_ylabel('Y')
        ax.set_zlabel('Field Coherence')

    plt.tight_layout()
    return fig
```

C.2 Cross-Domain Correlation Visualizations

C.2.1 Scale-Invariant Pattern Display

Multi-Scale Correlation Plot

python

```python
def visualize_cross_scale_correlations(quantum_data, neural_data, cosmic_data):
    """
    Visualize correlations across quantum, neural, and cosmic scales
    """
    fig, axes = plt.subplots(2, 2, figsize=(16, 12))

    # Normalize data to [0, 1]
    def normalize(data):
        return (data - np.min(data)) / (np.max(data) - np.min(data))

    quantum_norm = normalize(quantum_data['coherence'])
    neural_norm = normalize(neural_data['complexity'])
    cosmic_norm = normalize(cosmic_data['structure_formation'])

    # Time alignment
    max_len = max(len(quantum_norm), len(neural_norm), len(cosmic_norm))
    time_axis = np.linspace(0, 1, max_len)

    # Interpolate to common time axis
    quantum_interp = np.interp(time_axis,
                               np.linspace(0, 1, len(quantum_norm)),
                               quantum_norm)
    neural_interp = np.interp(time_axis,
                              np.linspace(0, 1, len(neural_norm)),
                              neural_norm)
    cosmic_interp = np.interp(time_axis,
                              np.linspace(0, 1, len(cosmic_norm)),
                              cosmic_norm)

    # Plot overlays
    ax1 = axes[0, 0]
    ax1.plot(time_axis, quantum_interp, 'b-', label='Quantum Coherence')
    ax1.plot(time_axis, neural_interp, 'g-', label='Neural Complexity')
    ax1.plot(time_axis, cosmic_interp, 'r-', label='Cosmic Structure')
    ax1.set_title('Cross-Scale Pattern Evolution')
    ax1.legend()
    ax1.grid(True, alpha=0.3)

    # Correlation matrix
    ax2 = axes[0, 1]
    corr_matrix = np.corrcoef([quantum_interp, neural_interp, cosmic_interp])
    im = ax2.imshow(corr_matrix, cmap='RdBu_r', vmin=-1, vmax=1)
    ax2.set_xticks([0, 1, 2])
```

```python
    ax2.set_yticks([0, 1, 2])
    ax2.set_xticklabels(['Quantum', 'Neural', 'Cosmic'])
    ax2.set_yticklabels(['Quantum', 'Neural', 'Cosmic'])
    plt.colorbar(im, ax=ax2)
    ax2.set_title('Cross-Scale Correlation Matrix')

    # Power spectrum analysis
    ax3 = axes[1, 0]
    for data, label, color in [(quantum_interp, 'Quantum', 'blue'),
                               (neural_interp, 'Neural', 'green'),
                               (cosmic_interp, 'Cosmic', 'red')]:
        f, psd = signal.welch(data)
        ax3.loglog(f, psd, color=color, label=label)
    ax3.set_xlabel('Frequency')
    ax3.set_ylabel('Power Spectral Density')
    ax3.legend()
    ax3.grid(True, alpha=0.3)
    ax3.set_title('Power Spectrum Comparison')

    # Phase relationship
    ax4 = axes[1, 1]
    ax4.scatter(quantum_interp, neural_interp, alpha=0.5, label='Quantum-Neural')
    ax4.scatter(neural_interp, cosmic_interp, alpha=0.5, label='Neural-Cosmic')
    ax4.scatter(quantum_interp, cosmic_interp, alpha=0.5, label='Quantum-Cosmic')
    ax4.set_xlabel('Normalized Value')
    ax4.set_ylabel('Normalized Value')
    ax4.legend()
    ax4.set_title('Phase Relationships')

    plt.tight_layout()
    return fig
```

C.2.2 Resonance Pattern Maps

Harmonic Resonance Visualization

python

```python
def visualize_resonance_patterns(resonance_data, frequency_bins):
    """
    Create detailed resonance pattern visualizations
    """
    fig = plt.figure(figsize=(18, 10))

    # 3D surface plot of resonance landscape
    ax1 = fig.add_subplot(2, 2, 1, projection='3d')

    X, Y = np.meshgrid(range(resonance_data.shape[0]),
                       range(resonance_data.shape[1]))

    surf = ax1.plot_surface(X, Y, resonance_data, cmap='magma')
    ax1.set_xlabel('Time')
    ax1.set_ylabel('Frequency Mode')
    ax1.set_zlabel('Resonance Amplitude')
    ax1.set_title('Resonance Landscape')

    # Frequency spectrum waterfall
    ax2 = fig.add_subplot(2, 2, 2)

    im = ax2.imshow(resonance_data.T, aspect='auto', cmap='inferno',
                    origin='lower', extent=[0, resonance_data.shape[0],
                                            frequency_bins[0], frequency_bins[-1]])
    ax2.set_xlabel('Time')
    ax2.set_ylabel('Frequency (Hz)')
    ax2.set_title('Resonance Spectrogram')
    plt.colorbar(im, ax=ax2)

    # Peak resonance tracking
    ax3 = fig.add_subplot(2, 2, 3)

    peak_indices = np.argmax(resonance_data, axis=1)
    peak_frequencies = frequency_bins[peak_indices]
    peak_amplitudes = np.max(resonance_data, axis=1)

    ax3.plot(peak_frequencies, 'b-', linewidth=2)
    ax3_twin = ax3.twinx()
    ax3_twin.plot(peak_amplitudes, 'r-', linewidth=2)

    ax3.set_xlabel('Time')
    ax3.set_ylabel('Peak Frequency (Hz)', color='blue')
    ax3_twin.set_ylabel('Peak Amplitude', color='red')
```

```
    ax3.set_title('Peak Resonance Tracking')

    # Resonance network diagram
    ax4 = fig.add_subplot(2, 2, 4)

    # Calculate resonance correlations
    corr_matrix = np.corrcoef(resonance_data.T)
    threshold = 0.7

    # Create network graph
    G = nx.Graph()
    n_modes = corr_matrix.shape[0]

    # Add nodes
    for i in range(n_modes):
        G.add_node(i, frequency=frequency_bins[i])

    # Add edges for strong correlations
    for i in range(n_modes):
        for j in range(i+1, n_modes):
            if corr_matrix[i, j] > threshold:
                G.add_edge(i, j, weight=corr_matrix[i, j])

    # Draw network
    pos = nx.spring_layout(G, k=0.5)
    nx.draw_networkx_nodes(G, pos, node_color='lightblue',
                           node_size=500, alpha=0.8, ax=ax4)

    edges = [(u, v) for (u, v, d) in G.edges(data=True)]
    weights = [G[u][v]['weight'] for u, v in edges]

    nx.draw_networkx_edges(G, pos, edgelist=edges, width=weights*5,
                           alpha=0.6, ax=ax4)

    ax4.set_title('Resonance Mode Network')
    ax4.axis('off')

    plt.tight_layout()
    return fig
```

C.3 Energy and Complexity Visualizations

C.3.1 Energy Translation Flow Diagrams

Sankey Diagram for Energy Translation

python

```
import plotly.graph_objects as go

def visualize_energy_translation(energy_flows):
    """
    Create Sankey diagram showing energy translation through system
    """
    # Define nodes
    nodes = dict(
        pad=15,
        thickness=20,
        line=dict(color="black", width=0.5),
        label=["Input Energy", "Intent Field", "Structure Formation",
               "Information Organization", "Coherence Maintenance",
               "Dissipated", "Stored Complexity", "Output Organization"],
        color=["blue", "green", "orange", "purple", "cyan",
               "red", "gold", "lightgreen"]
    )

    # Define links
    links = dict(
        source=[0, 1, 1, 1, 2, 3, 4, 5],
        target=[1, 2, 3, 4, 5, 6, 6, 7],
        value=energy_flows
    )

    fig = go.Figure(data=[go.Sankey(node=nodes, link=links)])

    fig.update_layout(title_text="Energy Translation Flow",
                      font_size=14,
                      width=1200,
                      height=800)

    return fig
```

Phase Space Visualization

python

```python
def visualize_energy_complexity_phase_space(energy_data, complexity_data,
                                            system_states):
    """
    Create phase space plot of energy vs complexity
    """
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 8))

    # Main phase space
    ax1.plot(energy_data, complexity_data, 'b-', alpha=0.6)

    # Color code by time
    times = np.arange(len(energy_data))
    scatter = ax1.scatter(energy_data, complexity_data,
                          c=times, cmap='viridis', s=20)

    # Mark special states
    for state_name, indices in system_states.items():
        for idx in indices:
            ax1.scatter(energy_data[idx], complexity_data[idx],
                        s=100, marker='*', color='red', zorder=10)
            ax1.annotate(state_name,
                         (energy_data[idx], complexity_data[idx]),
                         xytext=(5, 5), textcoords='offset points')

    ax1.set_xlabel('Energy Level')
    ax1.set_ylabel('Complexity')
    ax1.set_title('Energy-Complexity Phase Space')
    plt.colorbar(scatter, ax=ax1, label='Time')

    # Derivative analysis
    energy_derivative = np.gradient(energy_data)
    complexity_derivative = np.gradient(complexity_data)

    ax2.plot(energy_derivative, complexity_derivative, 'g-', alpha=0.6)
    ax2.scatter(energy_derivative, complexity_derivative,
                c=times, cmap='plasma', s=20)

    ax2.set_xlabel('dE/dt')
    ax2.set_ylabel('dC/dt')
    ax2.set_title('Phase Space Derivatives')
    ax2.axhline(y=0, color='k', linestyle='--', alpha=0.5)
    ax2.axvline(x=0, color='k', linestyle='--', alpha=0.5)
    plt.colorbar(scatter, ax=ax2, label='Time')
```

```python
    plt.tight_layout()
    return fig
```

C.4 Biological Development Visualizations

C.4.1 Neural Network Evolution

Network Growth Animation

python

```python
import matplotlib.animation as animation

def create_neural_development_animation(network_states, timestamps):
    """
    Create animation of neural network development
    """
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 7))

    def update(frame):
        ax1.clear()
        ax2.clear()

        state = network_states[frame]
        timestamp = timestamps[frame]

        # Network topology
        G = nx.from_numpy_array(state['adjacency'])
        pos = nx.spring_layout(G, k=0.5)

        # Color nodes by activation
        node_colors = [state['activations'][node] for node in G.nodes()]

        nx.draw_networkx_nodes(G, pos, node_color=node_colors,
                              cmap='coolwarm', node_size=300, ax=ax1)
        nx.draw_networkx_edges(G, pos, alpha=0.5, width=0.5, ax=ax1)

        ax1.set_title(f'Neural Network - Time: {timestamp}')
        ax1.axis('off')

        # Statistics
        connectivity = np.sum(state['adjacency']) / (len(state['adjacency'])**2)
        clustering = nx.average_clustering(G)
        path_length = nx.average_shortest_path_length(G) if nx.is_connected(G) else 0

        stats = [
            f"Connectivity: {connectivity:.3f}",
            f"Clustering: {clustering:.3f}",
            f"Avg Path Length: {path_length:.3f}",
            f"Total Nodes: {len(G.nodes())}",
            f"Total Edges: {len(G.edges())}"
        ]

        for i, stat in enumerate(stats):
```

```python
            ax2.text(0.05, 0.9-i*0.1, stat, transform=ax2.transAxes,
                    fontsize=12, verticalalignment='top')

        ax2.axis('off')
        ax2.set_title('Network Statistics')

    ani = animation.FuncAnimation(fig, update, frames=len(network_states),
                                 interval=200, blit=False)

    return ani
```

Synaptic Pruning Visualization

python

```python
def visualize_synaptic_pruning(synaptic_weights, pruning_thresholds):
    """
    Visualize synaptic pruning process
    """
    fig, axes = plt.subplots(2, 2, figsize=(15, 12))

    # Weight distribution over time
    ax1 = axes[0, 0]

    for t, weights in enumerate(synaptic_weights):
        ax1.hist(weights.flatten(), bins=50, alpha=0.5,
                label=f'Time {t}', density=True)

    ax1.set_xlabel('Synaptic Weight')
    ax1.set_ylabel('Probability Density')
    ax1.set_title('Synaptic Weight Distribution Evolution')
    ax1.legend()

    # Pruning process
    ax2 = axes[0, 1]

    pruned_counts = []
    total_synapses = []

    for t, (weights, threshold) in enumerate(zip(synaptic_weights, pruning_thresholds)
        pruned = np.sum(weights < threshold)
        total = len(weights.flatten())
        pruned_counts.append(pruned)
        total_synapses.append(total)

    times = range(len(synaptic_weights))
    ax2.plot(times, total_synapses, 'b-', label='Total Synapses')
    ax2.plot(times, pruned_counts, 'r-', label='Pruned Synapses')
    ax2.fill_between(times, 0, pruned_counts, alpha=0.3, color='red')

    ax2.set_xlabel('Development Time')
    ax2.set_ylabel('Synapse Count')
    ax2.set_title('Synaptic Pruning Process')
    ax2.legend()

    # Network efficiency
    ax3 = axes[1, 0]
```

```
efficiencies = []
for weights in synaptic_weights:
    # Simple efficiency metric: information transfer per connection
    efficiency = np.sum(weights**2) / np.count_nonzero(weights)
    efficiencies.append(efficiency)

ax3.plot(times, efficiencies, 'g-', linewidth=2)
ax3.set_xlabel('Development Time')
ax3.set_ylabel('Network Efficiency')
ax3.set_title('Efficiency Through Pruning')
ax3.grid(True, alpha=0.3)

# 3D connectivity visualization
ax4 = fig.add_subplot(2, 2, 4, projection='3d')

# Show final state connectivity
final_weights = synaptic_weights[-1]
n_neurons = int(np.sqrt(len(final_weights.flatten())))
adj_matrix = final_weights.reshape(n_neurons, n_neurons)

# Create 3D network
theta = np.linspace(0, 2*np.pi, n_neurons)
x = np.cos(theta)
y = np.sin(theta)
z = np.zeros_like(x)

# Draw connections
for i in range(n_neurons):
    for j in range(n_neurons):
        if adj_matrix[i, j] > pruning_thresholds[-1]:
            ax4.plot([x[i], x[j]], [y[i], y[j]], [z[i], z[j]],
                    'g-', alpha=min(adj_matrix[i, j], 1.0))

# Draw neurons
ax4.scatter(x, y, z, s=100, c='blue', alpha=0.8)

ax4.set_title('Final Network Architecture')
ax4.axis('off')

plt.tight_layout()
return fig
```

C.5 Cosmological Visualizations

C.5.1 Large-Scale Structure Evolution

Cosmic Web Visualization

python

```
def visualize_cosmic_web_evolution(density_fields, intent_fields, redshifts):
    """
    Visualize evolution of cosmic large-scale structure
    """
    fig = plt.figure(figsize=(20, 15))

    n_snapshots = len(density_fields)
    cols = 3
    rows = (n_snapshots + cols - 1) // cols

    for i, (density, intent, z) in enumerate(zip(density_fields, intent_fields, redshi
        ax = fig.add_subplot(rows, cols, i+1)

        # Create 2D projection
        density_2d = np.sum(density, axis=2)  # Sum along z-axis
        intent_2d = np.sum(intent, axis=2)

        # Plot density field
        im = ax.imshow(np.log10(density_2d + 1), cmap='inferno',
                       origin='lower', extent=[0, 100, 0, 100])

        # Overlay intent field contours
        contours = ax.contour(intent_2d, levels=5, colors='cyan',
                              linewidths=1, alpha=0.7)

        ax.set_title(f'z = {z:.1f}')
        ax.set_xlabel('Mpc/h')
        ax.set_ylabel('Mpc/h')

        if i == 0:
            plt.colorbar(im, ax=ax, label='log(ρ/ρ₀)')

    plt.tight_layout()
    return fig
```

Halo-Intent Field Correlation

python

```python
def visualize_halo_intent_correlation(halo_data, intent_field_data):
    """
    Visualize correlation between halo properties and intent field
    """
    fig, axes = plt.subplots(2, 2, figsize=(15, 12))

    # Halo mass vs local intent field strength
    ax1 = axes[0, 0]

    scatter = ax1.scatter(halo_data['masses'], halo_data['local_intent'],
                          c=halo_data['redshifts'], cmap='viridis',
                          alpha=0.6, s=halo_data['sizes']*10)

    ax1.set_xscale('log')
    ax1.set_xlabel('Halo Mass [M /h]')
    ax1.set_ylabel('Local Intent Field Strength')
    ax1.set_title('Halo Mass - Intent Correlation')
    plt.colorbar(scatter, ax=ax1, label='Redshift')

    # Concentration-Intent relationship
    ax2 = axes[0, 1]

    ax2.scatter(halo_data['concentrations'], halo_data['intent_coherence'],
               c=halo_data['masses'], cmap='plasma', alpha=0.6)

    ax2.set_xlabel('Halo Concentration')
    ax2.set_ylabel('Intent Field Coherence')
    ax2.set_title('Concentration - Coherence Correlation')
    plt.colorbar(scatter, ax=ax2, label='Halo Mass')

    # Radial profiles
    ax3 = axes[1, 0]

    radii = np.linspace(0, 5, 50)  # In virial radii
    for mass_bin in ['low', 'medium', 'high']:
        mask = halo_data['mass_bin'] == mass_bin
        avg_density = np.mean(halo_data['density_profiles'][mask], axis=0)
        avg_intent = np.mean(halo_data['intent_profiles'][mask], axis=0)

        ax3.plot(radii, avg_density, label=f'{mass_bin} mass - density')
        ax3.plot(radii, avg_intent*max(avg_density), '--',
                label=f'{mass_bin} mass - intent (scaled)')
```

```
ax3.set_xlabel('r/r_vir')
ax3.set_ylabel('ρ/ρ₀')
ax3.set_yscale('log')
ax3.legend()
ax3.set_title('Radial Profiles')

# Intent field topology
ax4 = axes[1, 1]

# Create network of high-coherence regions
coherence_threshold = np.percentile(intent_field_data['coherence'], 95)
high_coherence_mask = intent_field_data['coherence'] > coherence_threshold

# Find connected regions
from skimage.measure import label
labeled_regions = label(high_coherence_mask)

# Analyze topology
region_sizes = np.bincount(labeled_regions.ravel())[1:]  # Exclude background

ax4.hist(region_sizes, bins=50, alpha=0.7)
ax4.set_xscale('log')
ax4.set_yscale('log')
ax4.set_xlabel('Connected Region Size')
ax4.set_ylabel('Count')
ax4.set_title('Intent Field Topology')

plt.tight_layout()
return fig
```

C.6 Interactive Visualization Components

C.6.1 Dynamic Field Explorer

python

```python
import ipywidgets as widgets
from IPython.display import display, HTML

def create_interactive_field_explorer(field_data):
    """
    Create interactive 3D field explorer with widgets
    """
    # Create widgets
    time_slider = widgets.IntSlider(
        value=0,
        min=0,
        max=field_data.shape[-1]-1,
        step=1,
        description='Time:',
        continuous_update=False
    )

    component_selector = widgets.Dropdown(
        options=['Magnitude', 'X-component', 'Y-component', 'Z-component', 'Coherence'
        value='Magnitude',
        description='Field Component:'
    )

    slice_plane = widgets.Dropdown(
        options=['XY', 'XZ', 'YZ'],
        value='XY',
        description='Slice Plane:'
    )

    slice_position = widgets.IntSlider(
        value=field_data.shape[2]//2,
        min=0,
        max=field_data.shape[2]-1,
        step=1,
        description='Slice Position:',
        continuous_update=False
    )

    def update_plot(time, component, plane, position):
        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))

        # Extract slice data
        if plane == 'XY':
```

```python
        data_slice = field_data[:, :, position, :, time]
        extent = [0, field_data.shape[0], 0, field_data.shape[1]]
    elif plane == 'XZ':
        data_slice = field_data[:, position, :, :, time]
        extent = [0, field_data.shape[0], 0, field_data.shape[2]]
    else:  # YZ
        data_slice = field_data[position, :, :, :, time]
        extent = [0, field_data.shape[1], 0, field_data.shape[2]]

    # Select component
    if component == 'Magnitude':
        plot_data = np.sqrt(np.sum(data_slice**2, axis=2))
    elif component == 'X-component':
        plot_data = data_slice[:, :, 0]
    elif component == 'Y-component':
        plot_data = data_slice[:, :, 1]
    elif component == 'Z-component':
        plot_data = data_slice[:, :, 2]
    else:  # Coherence
        plot_data = data_slice[:, :, 3]

    # 2D field visualization
    im1 = ax1.imshow(plot_data, cmap='viridis', origin='lower',
                    extent=extent, aspect='equal')
    ax1.set_title(f'{component} - {plane} plane at {position}')
    plt.colorbar(im1, ax=ax1)

    # Field lines
    if component != 'Magnitude':
        if plane == 'XY':
            u, v = data_slice[:, :, 0], data_slice[:, :, 1]
        elif plane == 'XZ':
            u, v = data_slice[:, :, 0], data_slice[:, :, 2]
        else:
            u, v = data_slice[:, :, 1], data_slice[:, :, 2]

        ax2.streamplot(np.linspace(extent[0], extent[1], u.shape[0]),
                       np.linspace(extent[2], extent[3], u.shape[1]),
                       u.T, v.T, color='white', density=2)
        ax2.set_title('Field Lines')
        ax2.set_xlim(extent[:2])
        ax2.set_ylim(extent[2:])
    else:
        ax2.axis('off')
```

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

# Link widgets
interactive_plot = widgets.interactive(update_plot,
                                       time=time_slider,
                                       component=component_selector,
                                       plane=slice_plane,
                                       position=slice_position)

display(interactive_plot)
return interactive_plot
```

C.6.2 Real-Time Simulation Viewer

python

```python
class RealTimeSimulationViewer:
    """
    Real-time visualization of IntentSim simulations
    """
    def __init__(self, sim_engine):
        self.sim = sim_engine
        self.fig, self.axes = plt.subplots(2, 2, figsize=(15, 12))
        self.running = False

    def initialize_plots(self):
        """Set up initial plot structures"""
        # Field visualization
        self.field_im = self.axes[0, 0].imshow(np.zeros((50, 50)),
                                               cmap='viridis', animated=True)
        self.axes[0, 0].set_title('Intent Field')

        # Complexity evolution
        self.complexity_line, = self.axes[0, 1].plot([], [], 'b-')
        self.axes[0, 1].set_xlim(0, 1000)
        self.axes[0, 1].set_ylim(0, 3000)
        self.axes[0, 1].set_title('Complexity Evolution')

        # Energy tracking
        self.energy_line, = self.axes[1, 0].plot([], [], 'r-')
        self.axes[1, 0].set_xlim(0, 1000)
        self.axes[1, 0].set_ylim(0, 50)
        self.axes[1, 0].set_title('Energy Levels')

        # Coherence heatmap
        self.coherence_im = self.axes[1, 1].imshow(np.zeros((20, 20)),
                                                   cmap='hot', animated=True)
        self.axes[1, 1].set_title('Field Coherence')

        plt.tight_layout()

    def update_frame(self, frame):
        """Update visualization for current simulation step"""
        state = self.sim.get_current_state()

        # Update field visualization
        field_slice = state['field'][:, :, state['field'].shape[2]//2]
        self.field_im.set_array(np.abs(field_slice))
        self.field_im.set_clim(vmin=0, vmax=np.max(np.abs(field_slice)))
```

```
        # Update time series
        self.complexity_line.set_data(state['time_history'],
                                      state['complexity_history'])
        self.energy_line.set_data(state['time_history'],
                                  state['energy_history'])

        # Update coherence
        coherence_matrix = state['coherence_matrix']
        self.coherence_im.set_array(coherence_matrix)
        self.coherence_im.set_clim(vmin=0, vmax=1)

        # Adjust axis limits if needed
        if len(state['time_history']) > 0:
            self.axes[0, 1].set_xlim(0, max(state['time_history'][-1]+100, 1000))
            self.axes[1, 0].set_xlim(0, max(state['time_history'][-1]+100, 1000))

        return [self.field_im, self.complexity_line,
                self.energy_line, self.coherence_im]

    def start_visualization(self):
        """Start real-time visualization"""
        self.initialize_plots()

        ani = animation.FuncAnimation(self.fig, self.update_frame,
                                      frames=1000, interval=50, blit=True)

        # Add controls
        def toggle_pause(event):
            if self.running:
                ani.pause()
                self.running = False
            else:
                ani.resume()
                self.running = True

        def save_current_state(event):
            timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
            filename = f"simulation_state_{timestamp}.png"
            self.fig.savefig(filename, dpi=300, bbox_inches='tight')
            print(f"Saved: {filename}")

        # Add buttons
        ax_pause = plt.axes([0.81, 0.01, 0.08, 0.04])
```

```python
        ax_save = plt.axes([0.91, 0.01, 0.08, 0.04])

        btn_pause = Button(ax_pause, 'Pause/Resume')
        btn_save = Button(ax_save, 'Save State')

        btn_pause.on_clicked(toggle_pause)
        btn_save.on_clicked(save_current_state)

        self.running = True
        return ani
```

C.7 Publication-Quality Figures

C.7.1 Figure Generation Templates

python

```python
def create_publication_figure(data_dict, figure_type='multi_panel'):
    """
    Create publication-quality figures with consistent styling
    """
    # Set publication style
    plt.style.use('seaborn-paper')
    plt.rcParams.update({
        'font.size': 10,
        'axes.labelsize': 10,
        'axes.titlesize': 12,
        'xtick.labelsize': 8,
        'ytick.labelsize': 8,
        'legend.fontsize': 8,
        'figure.titlesize': 14,
        'font.family': 'DejaVu Sans',
        'mathtext.fontset': 'stix'
    })

    if figure_type == 'single_panel':
        fig, ax = plt.subplots(1, 1, figsize=(3.5, 3.5))
        return fig, ax

    elif figure_type == 'double_column':
        fig, axes = plt.subplots(2, 3, figsize=(7.2, 4.8))
        return fig, axes

    elif figure_type == 'full_page':
        fig, axes = plt.subplots(3, 4, figsize=(7.2, 9.6))
        return fig, axes

    else:  # Custom multi-panel
        fig = plt.figure(figsize=(7.2, 9.6))

        # Create custom grid
        gs = fig.add_gridspec(4, 4, hspace=0.3, wspace=0.3)

        # Main plot
        ax_main = fig.add_subplot(gs[0:2, 0:2])

        # Supporting plots
        ax_energy = fig.add_subplot(gs[0, 2:4])
        ax_coherence = fig.add_subplot(gs[1, 2:4])
        ax_spectrum = fig.add_subplot(gs[2, 0:2])
```

```python
        ax_correlation = fig.add_subplot(gs[2, 2:4])
        ax_phase = fig.add_subplot(gs[3, 0:4])

        return fig, (ax_main, ax_energy, ax_coherence,
                    ax_spectrum, ax_correlation, ax_phase)

def add_scalebar(ax, length, label, loc='lower right'):
    """Add scalebar to plot"""
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredSizeBar
    import matplotlib.font_manager as fm

    fontprops = fm.FontProperties(size=8)
    scalebar = AnchoredSizeBar(ax.transData,
                              length, label, loc,
                              pad=0.1,
                              color='black',
                              frameon=False,
                              size_vertical=0.5,
                              fontproperties=fontprops)

    ax.add_artist(scalebar)

def add_inset_zoom(ax, data, zoom_region, zoom_factor=2):
    """Add zoomed inset to main plot"""
    from mpl_toolkits.axes_grid1.inset_locator import inset_axes, mark_inset

    # Create inset
    axins = inset_axes(ax, width="30%", height="30%", loc='upper right')

    # Plot zoomed data
    axins.imshow(data[zoom_region[0]:zoom_region[1],
                     zoom_region[2]:zoom_region[3]],
                cmap=ax.images[0].get_cmap())

    # Mark the region
    mark_inset(ax, axins, loc1=2, loc2=4, fc="none", ec="red")

    # Remove ticks
    axins.set_xticks([])
    axins.set_yticks([])

    return axins
```

C.7.2 Animation Export Templates

python

```python
def export_animation(animation_data, output_format='mp4', dpi=150):
    """
    Export animations in various formats for publications
    """
    if output_format == 'mp4':
        # For papers with multimedia
        Writer = animation.writers['ffmpeg']
        writer = Writer(fps=15, metadata=dict(artist='IntentSim'), bitrate=1800)

        ani.save('simulation_evolution.mp4', writer=writer, dpi=dpi)

    elif output_format == 'gif':
        # For presentations
        ani.save('simulation_evolution.gif', writer='pillow', fps=10, dpi=dpi)

    elif output_format == 'frames':
        # For custom processing
        for i, frame in enumerate(animation_data):
            plt.figure(figsize=(8, 6))
            # Plot frame
            plt.savefig(f'frame_{i:04d}.png', dpi=dpi, bbox_inches='tight')
            plt.close()

    elif output_format == 'web':
        # For interactive online viewing
        from matplotlib.animation import HTMLWriter
        ani.save('simulation_interactive.html', writer=HTMLWriter(embed_frames=True))

def create_figure_legend(elements, title="", loc='upper right'):
    """
    Create comprehensive figure legend
    """
    legend_elements = []

    for element in elements:
        if element['type'] == 'line':
            legend_elements.append(Line2D([0], [0], color=element['color'],
                                          linestyle=element.get('linestyle', '-'),
                                          label=element['label']))
        elif element['type'] == 'marker':
            legend_elements.append(Line2D([0], [0], marker=element['marker'],
                                          color='w', markerfacecolor=element['color'],
                                          markersize=element.get('size', 10),
```

```python
                                            label=element['label']))
        elif element['type'] == 'patch':
            legend_elements.append(Patch(facecolor=element['color'],
                                           label=element['label']))

    if title:
        legend = plt.legend(handles=legend_elements, loc=loc, title=title)
    else:
        legend = plt.legend(handles=legend_elements, loc=loc)

    # Customize legend appearance
    legend.get_frame().set_facecolor('white')
    legend.get_frame().set_alpha(0.8)

    return legend
```

C.8 Visualization Gallery

C.8.1 Showcase Figures

The following templates demonstrate key visualizations for different audiences:

1. **Research Papers**: High-density multi-panel figures with detailed metrics
2. **Conference Presentations**: Clear, high-contrast visuals with minimal text
3. **Public Outreach**: Intuitive, aesthetically pleasing representations
4. **Technical Documentation**: Comprehensive plots with extensive annotations

C.8.2 Style Guidelines

- **Color Schemes**:
  - Scientific: viridis, plasma, inferno
  - Qualitative: Set3, Paired
  - Diverging: RdBu, coolwarm
- **Font Sizes**:
  - Journal articles: 8-10pt
  - Presentations: 12-14pt
  - Posters: 16-20pt
- **Resolution**:
  - Print: 300 DPI minimum

- Screen: 150 DPI
- Web: 96 DPI

C.8.3 Accessibility Considerations

All visualizations should include:

- High contrast options
- Colorblind-friendly palettes
- Alternative text descriptions
- Interactive features for screen readers
- Scalable vector graphics when possible

Notes

1. All visualization code is optimized for both static and interactive use
2. Templates are modular and can be combined as needed
3. Export functions support multiple output formats
4. Styling follows scientific publication standards
5. Performance considerations included for large datasets

For additional visualization examples and tutorials, see the IntentSim documentation at: https://docs.TheVoidIntent.com/IntentSim/visualization/