15 - Attractor

Mariana Emauz Valdetaro

02 April 2025

Table of contents

```python
# Complete Fibonacci-CFS Attractor Animation
# Enhanced version with organic, flowing ribbons
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.animation import FuncAnimation
from matplotlib.widgets import Button

class FibonacciCFSAttractor:
    def __init__(self, num_ribbons=8, steps=1800, dt=0.018):
        self.num_ribbons = num_ribbons
        self.steps = steps
        self.dt = dt
        self.phi = (1 + np.sqrt(5)) / 2  # Golden ratio
        self.trajectories = []
        self.fig = None
        self.ax = None
        self.lines = []
        self.connection_lines = []

        # Animation control
        self.is_paused = False
        self.current_frame = 0
```

```python
def advanced_dynamics(self, x, y, z, t, ribbon_offset):
    """Enhanced dynamics for smooth, organic flow"""
    r = np.sqrt(x**2 + y**2 + z**2)

    # Multi-level Fibonacci scaling
    level1 = int(np.log(r + 1) / np.log(self.phi)) if r > 0 else 0
    level2 = int(t * 0.08) % 12

    # Enhanced coherence with multiple harmonics
    coherence = (self.phi**(-level1 * 0.05) *
                 (1 + 0.25 * np.sin(level1 * 0.3 + t * 0.08)) *
                 (1 + 0.15 * np.cos(level2 * 0.2 + ribbon_offset)))

    # Multi-scale power law coupling
    coupling = (1.0 * (r + 0.5)**(-0.28) *
                (1 + 0.18 * np.cos(0.06 * t + ribbon_offset)) *
                (1 + 0.12 * np.sin(0.04 * t)))

    # Complex spiral feedback
    theta = np.arctan2(y, x) + 0.05 * t + ribbon_offset
    spiral_x = (0.4 * np.cos(theta + 0.08 * z) +
                0.15 * np.cos(1.8 * theta + 0.03 * t))
    spiral_y = (0.4 * np.sin(theta + 0.08 * z) +
                0.15 * np.sin(1.8 * theta + 0.03 * t))
    spiral_z = (0.18 * np.sin(0.12 * t + ribbon_offset) +
                0.08 * np.cos(0.06 * t + 1.5 * ribbon_offset))

    # Attractor dynamics
    dx = coupling * (self.phi * y - 0.75 * x + 0.08 * z) + spiral_x
    dy = coupling * (coherence * x - y - 0.04 * x * z) + spiral_y
    dz = coupling * (0.08 * x * y - 0.35 * coherence * z + 0.03 * x) + spiral_z
```

```
        return dx, dy, dz

    def generate_trajectory(self, ribbon_index):
        """Generate trajectory with organic initial conditions"""
        ribbon_offset = 2 * np.pi * ribbon_index / self.num_ribbons

        # Varied initial conditions for organic start
        r0 = 0.6 + 0.4 * np.sin(ribbon_offset + 0.5)
        theta0 = ribbon_offset + 0.3 * np.sin(1.5 * ribbon_offset)
        x = r0 * np.cos(theta0)
        y = r0 * np.sin(theta0)
        z = 0.4 * np.sin(ribbon_offset + 0.8) + 0.15 * ribbon_index / self.num_ribbons

        trajectory = []

        for step in range(self.steps):
            t = step * self.dt
            dx, dy, dz = self.advanced_dynamics(x, y, z, t, ribbon_offset)

            x += dx * self.dt
            y += dy * self.dt
            z += dz * self.dt

            trajectory.append([x, y, z])

        return np.array(trajectory)

    def generate_all_trajectories(self):
        """Generate all trajectories"""
        self.trajectories = []
        for i in range(self.num_ribbons):
```

```
            traj = self.generate_trajectory(i)
            self.trajectories.append(traj)
        print(f"Generated {len(self.trajectories)} trajectories")

    def setup_animation(self):
        """Setup animation with enhanced visuals for fading trails"""
        self.fig = plt.figure(figsize=(12, 9))
        self.ax = self.fig.add_subplot(111, projection='3d')

        # Enhanced dark background
        self.fig.patch.set_facecolor('black')
        self.ax.xaxis.pane.fill = False
        self.ax.yaxis.pane.fill = False
        self.ax.zaxis.pane.fill = False
        self.ax.xaxis.pane.set_edgecolor('black')
        self.ax.yaxis.pane.set_edgecolor('black')
        self.ax.zaxis.pane.set_edgecolor('black')
        self.ax.set_facecolor('black')
        self.ax.set_axis_off()

        # Enhanced color palette - using plasma for organic feel
        colors = plt.cm.plasma(np.linspace(0, 1, self.num_ribbons))

        # Initialize main ribbon lines (these will be used for the brightest trail segment)
        self.lines = []
        for i in range(self.num_ribbons):
            line, = self.ax.plot([], [], [], color=colors[i], alpha=0.9, linewidth=3.0)
            self.lines.append(line)

        # Initialize connection lines (minimal since we'll draw them dynamically)
        self.connection_lines = []
        for i in range(0, self.num_ribbons - 1, 2):
```

```
            conn_line, = self.ax.plot([], [], [], color='cyan', alpha=0.1, linewidth=1)
            self.connection_lines.append(conn_line)

        # Set limits
        self.ax.set_xlim([-6, 6])
        self.ax.set_ylim([-6, 6])
        self.ax.set_zlim([-3, 3])

        # Enhanced view angle
        self.ax.view_init(elev=25, azim=45)

        self.ax.set_title('Fibonacci-CFS Attractor: Fading Trails Animation',
                         color='white', fontsize=16, pad=20)

        return self.fig, self.ax

    def animate_frame(self, frame):
        """Enhanced animation function with fading trails"""
        if self.is_paused:
            return self.lines + self.connection_lines

        # Progressive trail length
        trail_length = 120 + int(20 * np.sin(frame * 0.02))
        max_points = min(frame * 2 + 100, self.steps)

        # Clear previous trails by redrawing the entire trail with segments
        for i, (traj, color) in enumerate(zip(self.trajectories, plt.cm.plasma(np.linspace(0, 
            if max_points > trail_length:
                start_idx = max_points - trail_length
                end_idx = max_points

                trail_data = traj[start_idx:end_idx]
```

```
        if len(trail_data) > 5:  # Need enough points for segments
            # Create multiple segments with decreasing opacity and thickness
            segment_length = len(trail_data) // 8  # Divide trail into 8 segments

            for seg in range(8):
                seg_start = seg * segment_length
                seg_end = min((seg + 1) * segment_length, len(trail_data))

                if seg_end > seg_start + 1:
                    segment_data = trail_data[seg_start:seg_end]

                    # Calculate fading: newest segments (higher seg) have higher opaci
                    fade_factor = (seg + 1) / 8.0  # 0.125 to 1.0
                    alpha = 0.2 + 0.7 * fade_factor  # 0.2 to 0.9
                    thickness = 1.0 + 2.0 * fade_factor  # 1.0 to 3.0

                    # Plot this segment
                    self.ax.plot(segment_data[:, 0], segment_data[:, 1], segment_data[
                               color=color, alpha=alpha, linewidth=thickness)

        # Update the main line object (for animation return)
        if len(trail_data) > 1:
            self.lines[i].set_data_3d(trail_data[-10:, 0], trail_data[-10:, 1], trail_
            self.lines[i].set_alpha(0.9)
            self.lines[i].set_linewidth(3.0)
# Update connection lines with fading trails
conn_idx = 0
for i in range(0, self.num_ribbons - 1, 2):
    if conn_idx < len(self.connection_lines) and max_points > 50:
        traj1 = self.trajectories[i]
        traj2 = self.trajectories[i + 1]
```

```
            # Create multiple connection segments with fading
            connection_points = range(max(0, max_points - 80), max_points, 10)
            for j, point_idx in enumerate(connection_points):
                if point_idx < len(traj1) and point_idx < len(traj2):
                    # Fade factor based on how recent the connection is
                    fade = (j + 1) / len(connection_points)
                    alpha = 0.02 + 0.08 * fade
                    thickness = 0.5 + 0.5 * fade

                    self.ax.plot([traj1[point_idx, 0], traj2[point_idx, 0]],
                                 [traj1[point_idx, 1], traj2[point_idx, 1]],
                                 [traj1[point_idx, 2], traj2[point_idx, 2]],
                                 color='cyan', alpha=alpha, linewidth=thickness)
            conn_idx += 1

        # Slowly rotate view for dynamic perspective
        self.ax.view_init(elev=25, azim=45 + frame * 0.3)

        return self.lines + self.connection_lines

    def toggle_pause(self, event):
        """Toggle pause/play"""
        self.is_paused = not self.is_paused

    def reset_animation(self, event):
        """Reset animation to beginning"""
        self.current_frame = 0

    def create_animation(self, frames=500, interval=70):
        """Create complete animation with controls"""
        print("Generating trajectories...")
        self.generate_all_trajectories()
```

```
print("Setting up animation...")
self.setup_animation()

# Add control buttons
plt.tight_layout()

ax_pause = plt.axes([0.02, 0.02, 0.08, 0.04])
pause_button = Button(ax_pause, 'Pause', color='lightblue', hovercolor='blue')
pause_button.on_clicked(self.toggle_pause)

ax_reset = plt.axes([0.12, 0.02, 0.08, 0.04])
reset_button = Button(ax_reset, 'Reset', color='lightgreen', hovercolor='green')
reset_button.on_clicked(self.reset_animation)

# Add zoom functionality
def on_scroll(event):
    if event.inaxes == self.ax:
        xlim = self.ax.get_xlim()
        ylim = self.ax.get_ylim()
        zlim = self.ax.get_zlim()

        zoom_factor = 1.1 if event.step < 0 else 1/1.1

        x_center = (xlim[0] + xlim[1]) / 2
        y_center = (ylim[0] + ylim[1]) / 2
        z_center = (zlim[0] + zlim[1]) / 2

        x_range = (xlim[1] - xlim[0]) * zoom_factor / 2
        y_range = (ylim[1] - ylim[0]) * zoom_factor / 2
        z_range = (zlim[1] - zlim[0]) * zoom_factor / 2
```

```
            self.ax.set_xlim(x_center - x_range, x_center + x_range)
            self.ax.set_ylim(y_center - y_range, y_center + y_range)
            self.ax.set_zlim(z_center - z_range, z_center + z_range)

            self.fig.canvas.draw()

        self.fig.canvas.mpl_connect('scroll_event', on_scroll)

        print("Creating animation...")
        anim = FuncAnimation(
            self.fig, self.animate_frame, frames=frames,
            interval=interval, blit=False, repeat=True
        )

        return anim

# Create and run the enhanced animation
print("Creating enhanced Fibonacci-CFS attractor...")
attractor = FibonacciCFSAttractor(num_ribbons=8, steps=1800, dt=0.018)
anim = attractor.create_animation(frames=500, interval=70)

plt.show()

print("\nEnhanced animation features:")
print("- Organic flowing ribbons with Fibonacci scaling")
print("- Power law coupling for scale invariance")
print("- Dynamic translucency and plasma color palette")
print("- Smooth connection ribbons for depth")
print("- Interactive controls: play/pause, reset, zoom")
print("- Mathematical rigor demonstrating CFS framework")
```

```
Creating enhanced Fibonacci-CFS attractor...
Generating trajectories...
```

```
Generated 8 trajectories
Setting up animation...
Creating animation...

Enhanced animation features:
- Organic flowing ribbons with Fibonacci scaling
- Power law coupling for scale invariance
- Dynamic translucency and plasma color palette
- Smooth connection ribbons for depth
- Interactive controls: play/pause, reset, zoom
- Mathematical rigor demonstrating CFS framework
```

```python
# Fibonacci-CFS Attractor with Native Plotly Animation for Quarto
import numpy as np
import plotly.graph_objects as go
import plotly.express as px
from plotly.subplots import make_subplots

class FibonacciCFSAttractorPlotly:
    def __init__(self, num_ribbons=8, steps=1200, dt=0.018):
        self.num_ribbons = num_ribbons
        self.steps = steps
        self.dt = dt
        self.phi = (1 + np.sqrt(5)) / 2  # Golden ratio
        self.trajectories = []

    def advanced_dynamics(self, x, y, z, t, ribbon_offset):
        """Enhanced dynamics for smooth, organic flow"""
        r = np.sqrt(x**2 + y**2 + z**2)

        # Multi-level Fibonacci scaling
        level1 = int(np.log(r + 1) / np.log(self.phi)) if r > 0 else 0
        level2 = int(t * 0.08) % 12

        # Enhanced coherence with multiple harmonics
        coherence = (self.phi**(-level1 * 0.05) *
                     (1 + 0.25 * np.sin(level1 * 0.3 + t * 0.08)) *
                     (1 + 0.15 * np.cos(level2 * 0.2 + ribbon_offset)))

        # Multi-scale power law coupling
        coupling = (1.0 * (r + 0.5)**(-0.28) *
                    (1 + 0.18 * np.cos(0.06 * t + ribbon_offset)) *
                    (1 + 0.12 * np.sin(0.04 * t)))
```

```
        # Complex spiral feedback
        theta = np.arctan2(y, x) + 0.05 * t + ribbon_offset
        spiral_x = (0.4 * np.cos(theta + 0.08 * z) +
                    0.15 * np.cos(1.8 * theta + 0.03 * t))
        spiral_y = (0.4 * np.sin(theta + 0.08 * z) +
                    0.15 * np.sin(1.8 * theta + 0.03 * t))
        spiral_z = (0.18 * np.sin(0.12 * t + ribbon_offset) +
                    0.08 * np.cos(0.06 * t + 1.5 * ribbon_offset))

        # Attractor dynamics
        dx = coupling * (self.phi * y - 0.75 * x + 0.08 * z) + spiral_x
        dy = coupling * (coherence * x - y - 0.04 * x * z) + spiral_y
        dz = coupling * (0.08 * x * y - 0.35 * coherence * z + 0.03 * x) + spiral_z

        return dx, dy, dz

    def generate_trajectory(self, ribbon_index):
        """Generate trajectory with organic initial conditions"""
        ribbon_offset = 2 * np.pi * ribbon_index / self.num_ribbons

        # Varied initial conditions for organic start
        r0 = 0.6 + 0.4 * np.sin(ribbon_offset + 0.5)
        theta0 = ribbon_offset + 0.3 * np.sin(1.5 * ribbon_offset)
        x = r0 * np.cos(theta0)
        y = r0 * np.sin(theta0)
        z = 0.4 * np.sin(ribbon_offset + 0.8) + 0.15 * ribbon_index / self.num_ribbons

        trajectory = []

        for step in range(self.steps):
            t = step * self.dt
            dx, dy, dz = self.advanced_dynamics(x, y, z, t, ribbon_offset)
```

```
            x += dx * self.dt
            y += dy * self.dt
            z += dz * self.dt

            trajectory.append([x, y, z])

        return np.array(trajectory)

    def generate_all_trajectories(self):
        """Generate all trajectories"""
        self.trajectories = []
        for i in range(self.num_ribbons):
            traj = self.generate_trajectory(i)
            self.trajectories.append(traj)
        return self.trajectories

    def create_animated_figure(self):
        """Create animated figure with native Plotly animation"""
        print("Generating trajectories...")
        self.generate_all_trajectories()

        # Create color palette
        colors = px.colors.qualitative.Set3[:self.num_ribbons]

        # Prepare animation frames
        frames = []
        animation_steps = 150  # Number of animation frames
        trail_length = 120

        for frame_idx in range(animation_steps):
            frame_data = []
```

```
max_points = min(frame_idx * 8 + 100, self.steps)

# Add ribbon trails for this frame
for ribbon_idx, (traj, color) in enumerate(zip(self.trajectories, colors)):
    if max_points > trail_length:
        start_idx = max_points - trail_length
        end_idx = max_points
        trail_data = traj[start_idx:end_idx]

        # Create multiple segments with fading effect
        segment_length = max(1, len(trail_data) // 6)

        for seg in range(6):
            seg_start = seg * segment_length
            seg_end = min((seg + 1) * segment_length, len(trail_data))

            if seg_end > seg_start:
                segment_data = trail_data[seg_start:seg_end]

                # Fading parameters
                fade_factor = (seg + 1) / 6.0
                opacity = 0.2 + 0.6 * fade_factor
                line_width = 2 + 4 * fade_factor

                # Add segment trace
                frame_data.append(
                    go.Scatter3d(
                        x=segment_data[:, 0],
                        y=segment_data[:, 1],
                        z=segment_data[:, 2],
                        mode='lines',
                        line=dict(color=color, width=line_width),
```

```
                        opacity=opacity,
                        showlegend=False,
                        name=f'Ribbon {ribbon_idx+1}'
                    )
                )

        # Add current position marker
        if max_points > 0 and max_points <= len(traj):
            current_point = traj[max_points-1]
            frame_data.append(
                go.Scatter3d(
                    x=[current_point[0]],
                    y=[current_point[1]],
                    z=[current_point[2]],
                    mode='markers',
                    marker=dict(size=10, color=color, opacity=1.0),
                    showlegend=False,
                    name=f'Current {ribbon_idx+1}'
                )
            )

    # Add connection ribbons
    for i in range(0, self.num_ribbons - 1, 2):
        if i + 1 < len(self.trajectories) and max_points > 30:
            traj1 = self.trajectories[i]
            traj2 = self.trajectories[i + 1]

            # Sample connection points
            connection_indices = range(max(0, max_points - 80), max_points, 20)
            for point_idx in connection_indices:
                if point_idx < len(traj1) and point_idx < len(traj2):
                    frame_data.append(
```

```python
                    go.Scatter3d(
                        x=[traj1[point_idx, 0], traj2[point_idx, 0]],
                        y=[traj1[point_idx, 1], traj2[point_idx, 1]],
                        z=[traj1[point_idx, 2], traj2[point_idx, 2]],
                        mode='lines',
                        line=dict(color='rgba(0, 200, 200, 0.4)', width=3),
                        showlegend=False,
                        name='Connection'
                    )
                )

    # Create frame
    frames.append(go.Frame(data=frame_data, name=str(frame_idx)))

# Create initial figure with first frame data
fig = go.Figure(data=frames[0].data, frames=frames)

# Update layout
fig.update_layout(
    title=dict(
        text="Fibonacci-CFS Attractor: Interactive Animation",
        x=0.5,
        font=dict(size=18, color='#2c3e50')
    ),
    scene=dict(
        xaxis=dict(
            range=[-6, 6],
            showgrid=True,
            gridcolor='lightgray',
            title='X'
        ),
        yaxis=dict(
```

```
            range=[-6, 6],
            showgrid=True,
            gridcolor='lightgray',
            title='Y'
        ),
        zaxis=dict(
            range=[-3, 3],
            showgrid=True,
            gridcolor='lightgray',
            title='Z'
        ),
        bgcolor='white',
        camera=dict(
            eye=dict(x=1.8, y=1.8, z=1.2),
            center=dict(x=0, y=0, z=0)
        ),
        aspectmode='cube'
    ),
    updatemenus=[{
        'type': 'buttons',
        'showactive': False,
        'buttons': [
            {
                'label': 'Play',
                'method': 'animate',
                'args': [None, {
                    'frame': {'duration': 100, 'redraw': True},
                    'fromcurrent': True,
                    'transition': {'duration': 50}
                }]
            },
            {
```

```
                'label': 'Pause',
                'method': 'animate',
                'args': [[None], {
                    'frame': {'duration': 0, 'redraw': False},
                    'mode': 'immediate',
                    'transition': {'duration': 0}
                }]
            },
            {
                'label': 'Reset',
                'method': 'animate',
                'args': [['0'], {
                    'frame': {'duration': 0, 'redraw': True},
                    'mode': 'immediate',
                    'transition': {'duration': 0}
                }]
            }
        ],
        'direction': 'left',
        'pad': {'r': 10, 't': 87},
        'x': 0.1,
        'xanchor': 'right',
        'y': 0,
        'yanchor': 'top'
    }],
    sliders=[{
        'steps': [
            {
                'args': [[str(k)], {
                    'frame': {'duration': 0, 'redraw': True},
                    'mode': 'immediate',
                    'transition': {'duration': 0}
```

```
                        }],
                        'label': str(k),
                        'method': 'animate'
                    } for k in range(len(frames))
                ],
                'active': 0,
                'yanchor': 'top',
                'xanchor': 'left',
                'currentvalue': {
                    'font': {'size': 16},
                    'prefix': 'Frame: ',
                    'visible': True,
                    'xanchor': 'right'
                },
                'transition': {'duration': 0},
                'x': 0.1,
                'y': 0,
                'len': 0.9
            }],
            width=900,
            height=700,
            margin=dict(l=0, r=0, t=50, b=100),
            paper_bgcolor='white',
            plot_bgcolor='white'
        )

        return fig

# Create and display the animation
print("Creating Fibonacci-CFS attractor animation...")
attractor = FibonacciCFSAttractorPlotly(num_ribbons=8, steps=1200, dt=0.018)
fig = attractor.create_animated_figure()
```

```
# Display the figure
fig.show()

print("\nInteractive Animation Features:")
print("  Play/Pause/Reset buttons")
print("  Frame slider for manual control")
print("  3D zoom, pan, and rotate")
print("  Fading trail visualization")
print("  Connection ribbons between trajectories")
print("  Mathematical rigor with Fibonacci-CFS framework")
print("  Native Quarto/Jupyter compatibility")
```

```
Creating Fibonacci-CFS attractor animation...
Generating trajectories...

Interactive Animation Features:
  Play/Pause/Reset buttons
  Frame slider for manual control
  3D zoom, pan, and rotate
  Fading trail visualization
  Connection ribbons between trajectories
  Mathematical rigor with Fibonacci-CFS framework
  Native Quarto/Jupyter compatibility
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): text/html
```

```
# Fibonacci-CFS Attractor with Plotly - Direct Port from Working Matplotlib Version
import numpy as np
import plotly.graph_objects as go
import plotly.express as px
from plotly.subplots import make_subplots
```

```python
class FibonacciCFSAttractorPlotly:
    def __init__(self, num_ribbons=8, steps=1800, dt=0.018):
        self.num_ribbons = num_ribbons
        self.steps = steps
        self.dt = dt
        self.phi = (1 + np.sqrt(5)) / 2  # Golden ratio
        self.trajectories = []

    def advanced_dynamics(self, x, y, z, t, ribbon_offset):
        """Enhanced dynamics for smooth, organic flow - EXACT COPY from working version"""
        r = np.sqrt(x**2 + y**2 + z**2)

        # Multi-level Fibonacci scaling
        level1 = int(np.log(r + 1) / np.log(self.phi)) if r > 0 else 0
        level2 = int(t * 0.08) % 12

        # Enhanced coherence with multiple harmonics
        coherence = (self.phi**(-level1 * 0.05) *
                    (1 + 0.25 * np.sin(level1 * 0.3 + t * 0.08)) *
                    (1 + 0.15 * np.cos(level2 * 0.2 + ribbon_offset)))

        # Multi-scale power law coupling
        coupling = (1.0 * (r + 0.5)**(-0.28) *
                   (1 + 0.18 * np.cos(0.06 * t + ribbon_offset)) *
                   (1 + 0.12 * np.sin(0.04 * t)))

        # Complex spiral feedback
        theta = np.arctan2(y, x) + 0.05 * t + ribbon_offset
        spiral_x = (0.4 * np.cos(theta + 0.08 * z) +
                   0.15 * np.cos(1.8 * theta + 0.03 * t))
        spiral_y = (0.4 * np.sin(theta + 0.08 * z) +
                   0.15 * np.sin(1.8 * theta + 0.03 * t))
```

```
        spiral_z = (0.18 * np.sin(0.12 * t + ribbon_offset) +
                    0.08 * np.cos(0.06 * t + 1.5 * ribbon_offset))

        # Attractor dynamics
        dx = coupling * (self.phi * y - 0.75 * x + 0.08 * z) + spiral_x
        dy = coupling * (coherence * x - y - 0.04 * x * z) + spiral_y
        dz = coupling * (0.08 * x * y - 0.35 * coherence * z + 0.03 * x) + spiral_z

        return dx, dy, dz

    def generate_trajectory(self, ribbon_index):
        """Generate trajectory with organic initial conditions - EXACT COPY"""
        ribbon_offset = 2 * np.pi * ribbon_index / self.num_ribbons

        # Varied initial conditions for organic start
        r0 = 0.6 + 0.4 * np.sin(ribbon_offset + 0.5)
        theta0 = ribbon_offset + 0.3 * np.sin(1.5 * ribbon_offset)
        x = r0 * np.cos(theta0)
        y = r0 * np.sin(theta0)
        z = 0.4 * np.sin(ribbon_offset + 0.8) + 0.15 * ribbon_index / self.num_ribbons

        trajectory = []

        for step in range(self.steps):
            t = step * self.dt
            dx, dy, dz = self.advanced_dynamics(x, y, z, t, ribbon_offset)

            x += dx * self.dt
            y += dy * self.dt
            z += dz * self.dt

            trajectory.append([x, y, z])
```

```python
        return np.array(trajectory)

    def generate_all_trajectories(self):
        """Generate all trajectories - EXACT COPY"""
        self.trajectories = []
        for i in range(self.num_ribbons):
            traj = self.generate_trajectory(i)
            self.trajectories.append(traj)
        print(f"Generated {len(self.trajectories)} trajectories")

    def create_frame_data(self, frame_num):
        """Create Plotly traces for a specific frame - mimicking matplotlib animation logic"""
        traces = []

        # Progressive trail length - same logic as matplotlib version
        trail_length = 120 + int(20 * np.sin(frame_num * 0.02))
        max_points = min(frame_num * 2 + 100, self.steps)

        # Create plasma colors matching matplotlib version
        plasma_colors = [
            '#0d0887', '#46039f', '#7201a8', '#9c179e', '#bd3786',
            '#d8576b', '#ed7953', '#fb9f3a', '#fdca26', '#f0f921'
        ]
        colors = [plasma_colors[int(i * (len(plasma_colors)-1) / (self.num_ribbons-1))]
                  for i in range(self.num_ribbons)]

        # Generate ribbon trails with fading effect - SAME LOGIC as matplotlib
        for i, (traj, color) in enumerate(zip(self.trajectories, colors)):
            if max_points > trail_length:
                start_idx = max_points - trail_length
                end_idx = max_points
```

```python
trail_data = traj[start_idx:end_idx]
if len(trail_data) > 5:  # Need enough points for segments
    # Create multiple segments with decreasing opacity and thickness
    segment_length = len(trail_data) // 8  # Divide trail into 8 segments

    for seg in range(8):
        seg_start = seg * segment_length
        seg_end = min((seg + 1) * segment_length, len(trail_data))

        if seg_end > seg_start + 1:
            segment_data = trail_data[seg_start:seg_end]

            # Calculate fading: newest segments (higher seg) have higher opaci
            fade_factor = (seg + 1) / 8.0  # 0.125 to 1.0
            alpha = 0.2 + 0.7 * fade_factor  # 0.2 to 0.9
            thickness = 1.0 + 2.0 * fade_factor  # 1.0 to 3.0

            # Create Plotly trace for this segment
            trace = go.Scatter3d(
                x=segment_data[:, 0],
                y=segment_data[:, 1],
                z=segment_data[:, 2],
                mode='lines',
                line=dict(
                    color=color,
                    width=thickness * 2,  # Plotly uses different scale
                ),
                opacity=alpha,
                showlegend=False,
                name=f'Ribbon {i+1} Segment {seg}'
            )
```

```
                    traces.append(trace)

    # Add connection lines with fading trails - SAME LOGIC as matplotlib
    for i in range(0, self.num_ribbons - 1, 2):
        if max_points > 50 and i + 1 < len(self.trajectories):
            traj1 = self.trajectories[i]
            traj2 = self.trajectories[i + 1]

            # Create multiple connection segments with fading
            connection_points = range(max(0, max_points - 80), max_points, 10)
            for j, point_idx in enumerate(connection_points):
                if point_idx < len(traj1) and point_idx < len(traj2):
                    # Fade factor based on how recent the connection is
                    fade = (j + 1) / len(connection_points)
                    alpha = 0.02 + 0.08 * fade
                    thickness = 0.5 + 0.5 * fade

                    conn_trace = go.Scatter3d(
                        x=[traj1[point_idx, 0], traj2[point_idx, 0]],
                        y=[traj1[point_idx, 1], traj2[point_idx, 1]],
                        z=[traj1[point_idx, 2], traj2[point_idx, 2]],
                        mode='lines',
                        line=dict(
                            color='cyan',
                            width=thickness * 2
                        ),
                        opacity=alpha,
                        showlegend=False,
                        name=f'Connection {i}-{i+1}'
                    )
                    traces.append(conn_trace)
```

```
        return traces

    def create_animated_figure(self):
        """Create animated Plotly figure with same visual style as matplotlib version"""
        print("Generating trajectories...")
        self.generate_all_trajectories()

        # Create animation frames
        frames = []
        animation_steps = 200  # Number of animation frames

        print("Creating animation frames...")
        for frame_idx in range(animation_steps):
            frame_data = self.create_frame_data(frame_idx)
            frames.append(go.Frame(data=frame_data, name=str(frame_idx)))

        # Create initial figure with first frame
        fig = go.Figure(data=frames[0].data, frames=frames)

        # Update layout to match matplotlib dark theme and view
        fig.update_layout(
            title=dict(
                text="Fibonacci-CFS Attractor: Organic Flow with Fading Trails",
                x=0.5,
                font=dict(size=18, color='white')
            ),
            scene=dict(
                xaxis=dict(
                    range=[-6, 6],
                    showgrid=False,
                    showline=False,
                    zeroline=False,
```

```
            showticklabels=False,
            title=''
        ),
        yaxis=dict(
            range=[-6, 6],
            showgrid=False,
            showline=False,
            zeroline=False,
            showticklabels=False,
            title=''
        ),
        zaxis=dict(
            range=[-3, 3],
            showgrid=False,
            showline=False,
            zeroline=False,
            showticklabels=False,
            title=''
        ),
        bgcolor='black',  # Dark background like matplotlib version
        camera=dict(
            eye=dict(x=1.8, y=1.8, z=1.2),
            center=dict(x=0, y=0, z=0)
        ),
        aspectmode='cube'
    ),
    updatemenus=[{
        'type': 'buttons',
        'showactive': False,
        'buttons': [
            {
                'label': '  Play',
```

```
            'method': 'animate',
            'args': [None, {
                'frame': {'duration': 70, 'redraw': True},  # Same as matplotlib in
                'fromcurrent': True,
                'transition': {'duration': 30}
            }]
        },
        {
            'label': '  Pause',
            'method': 'animate',
            'args': [[None], {
                'frame': {'duration': 0, 'redraw': False},
                'mode': 'immediate',
                'transition': {'duration': 0}
            }]
        },
        {
            'label': '  Reset',
            'method': 'animate',
            'args': [['0'], {
                'frame': {'duration': 0, 'redraw': True},
                'mode': 'immediate',
                'transition': {'duration': 0}
            }]
        }
    ],
    'direction': 'left',
    'pad': {'r': 10, 't': 87},
    'x': 0.1,
    'xanchor': 'right',
    'y': 0,
    'yanchor': 'top',
```

```
        'bgcolor': 'rgba(0,0,0,0.5)',
        'bordercolor': 'white',
        'font': {'color': 'white'}
    }],
    sliders=[{
        'steps': [
            {
                'args': [[str(k)], {
                    'frame': {'duration': 0, 'redraw': True},
                    'mode': 'immediate',
                    'transition': {'duration': 0}
                }],
                'label': str(k),
                'method': 'animate'
            } for k in range(len(frames))
        ],
        'active': 0,
        'yanchor': 'top',
        'xanchor': 'left',
        'currentvalue': {
            'font': {'size': 16, 'color': 'white'},
            'prefix': 'Frame: ',
            'visible': True,
            'xanchor': 'right'
        },
        'transition': {'duration': 0},
        'x': 0.1,
        'y': 0,
        'len': 0.9,
        'bgcolor': 'rgba(0,0,0,0.5)',
        'bordercolor': 'white'
    }],
```

```python
            width=900,
            height=700,
            margin=dict(l=0, r=0, t=50, b=100),
            paper_bgcolor='black',  # Black background like matplotlib
            plot_bgcolor='black'
        )

        return fig

# Create and display the animation
print("Creating enhanced Fibonacci-CFS attractor with Plotly...")
attractor = FibonacciCFSAttractorPlotly(num_ribbons=8, steps=1800, dt=0.018)
fig = attractor.create_animated_figure()

# Display the figure
fig.show()

print("\nPlotly Animation Features (matching matplotlib version):")
print("  EXACT same mathematical dynamics")
print("  Organic flowing ribbons with Fibonacci scaling")
print("  Fading trail visualization with 8 segments per ribbon")
print("  Connection ribbons between trajectories")
print("  Dark background matching matplotlib aesthetic")
print("  Play/Pause/Reset controls")
print("  Frame slider for manual control")
print("  Native Quarto compatibility")
print("  Mathematical rigor demonstrating CFS framework")
```

```
Creating enhanced Fibonacci-CFS attractor with Plotly...
Generating trajectories...
Generated 8 trajectories
Creating animation frames...
```

```
Plotly Animation Features (matching matplotlib version):
  EXACT same mathematical dynamics
  Organic flowing ribbons with Fibonacci scaling
  Fading trail visualization with 8 segments per ribbon
  Connection ribbons between trajectories
  Dark background matching matplotlib aesthetic
  Play/Pause/Reset controls
  Frame slider for manual control
  Native Quarto compatibility
  Mathematical rigor demonstrating CFS framework
```

```
Unable to display output for mime type(s): text/html
```