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

```
Mounted at /content/drive
```

```
import csv
import numpy as np
import pandas as pd
import math
import time

```

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.mplot3d.art3d import Poly3DCollection

```

```
import os
import torch
import numpy as np
import pickle
import logging
import random
from tqdm import tqdm
#from torch_geometric.data import Dataset
```

Read data

```
import sys
sys.path.append('/content/drive/MyDrive/action/data/processed/mmr_kp')
```

```
filename = "/content/drive/MyDrive/action/data/processed/mmr_kp/data.pkl"
```

```
import os
import numpy as np
import pandas as pd
import pickle
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

class MMRDataAnalyzer:
    def __init__(self, data_path):
        """Initialize analyzer with data path"""
```

```
        self.data_path = data_path
        self.data = self.load_processed_data()
        self.train_data = self.data['train']
        self.full_df = None

    def load_processed_data(self):
        """Load processed MMR data from pickle file"""
        with open(self.data_path, 'rb') as f:
            return pickle.load(f)

    def get_frame_data(self, data_point):
        """Convert a single data point to DataFrame"""
        x = data_point['new_x']
        df = pd.DataFrame({
            'frame_num': np.zeros(len(x)),
            'x': x[:, 0],
            'y': x[:, 1],
            'z': x[:, 2]
        })
        if 'cluster_labels' in data_point:
            df['cluster'] = data_point['cluster_labels']
        return df

    def analyze_dataset(self):
        """Analyze entire dataset"""
        all_points = []
        all_frames = []
        frame_count = 0

        for data_point in self.train_data:
            x = data_point['new_x']
            num_points = len(x)
            all_points.append(x)
            all_frames.extend([frame_count] * num_points)
            frame_count += 1

        all_points = np.concatenate(all_points, axis=0)
        self.full_df = pd.DataFrame({
            'frame_num': all_frames,
            'x': all_points[:, 0],
            'y': all_points[:, 1],
            'z': all_points[:, 2]
        })
        return self.full_df

    def visualize_frame(self, frame_idx=0, title="Point Cloud Visualization'
        """Visualize a single frame of point cloud data"""
        data_point = self.train_data[frame_idx]
        x = data_point['new_x']

        fig = plt.figure(figsize=(12, 8))
```

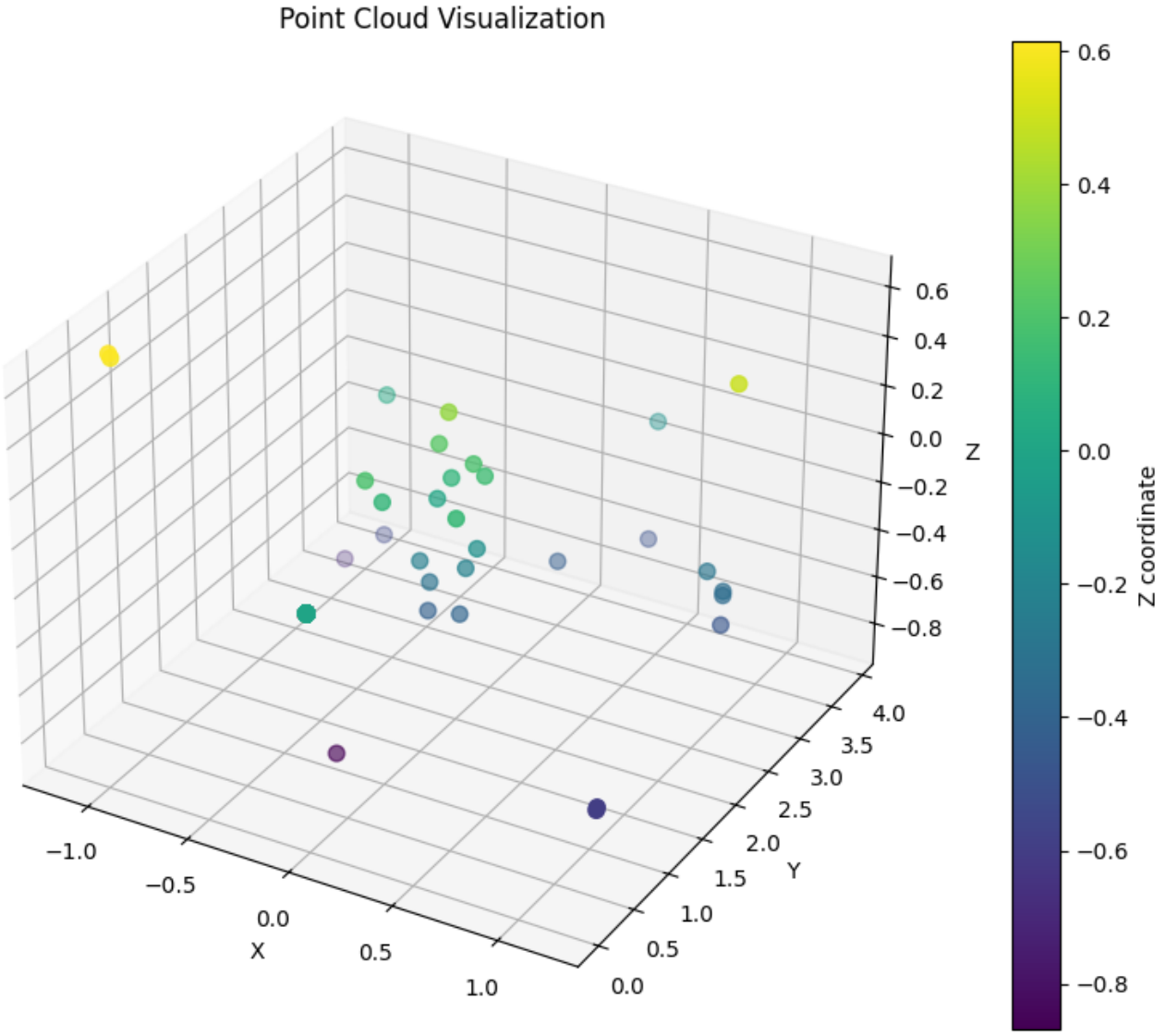
```
        ax = fig.add_subplot(111, projection='3d')
        scatter = ax.scatter(x[:, 0], x[:, 1], x[:, 2],
                             c=x[:, 2],
                             cmap='viridis',
                             marker='o',
                             s=50)

        ax.set_xlabel('X')
        ax.set_ylabel('Y')
        ax.set_zlabel('Z')
        ax.set_title(title)
        plt.colorbar(scatter, label='Z coordinate')
        plt.show()

    def analyze_by_action(self):
        """Analyze data grouped by action labels"""
        if 'y' not in self.train_data[0]:
            return None

        action_data = {}
        action_labels = [d['y'] for d in self.train_data]

        for data_point, action in zip(self.train_data, action_labels):
            if action not in action_data:
                action_data[action] = []
            action_data[action].append(data_point['new_x'])

        stats = {}
        for action, points in action_data.items():
            points = np.concatenate(points, axis=0)
            df = pd.DataFrame({
                'x': points[:, 0],
                'y': points[:, 1],
                'z': points[:, 2]
            })
            stats[action] = df.describe()
        return stats

    def plot_coordinate_distributions(self):
        """Plot distributions of x, y, z coordinates"""
        if self.full_df is None:
            self.analyze_dataset()

        fig, axes = plt.subplots(1, 3, figsize=(15, 5))

        for i, coord in enumerate(['x', 'y', 'z']):
            axes[i].hist(self.full_df[coord], bins=50)
            axes[i].set_title(f'{coord.upper()} Distribution')
            axes[i].set_xlabel(f'{coord.upper()} Coordinate')
            axes[i].set_ylabel('Frequency')

```

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

    def print_detailed_statistics(self):
        """Print detailed statistics for each coordinate"""
        if self.full_df is None:
            self.analyze_dataset()

        # Percentile analysis
        for coord in ['x', 'y', 'z']:
            print(f"\nDetailed {coord.upper()} coordinate analysis:")
            percentiles = [0, 1, 5, 25, 50, 75, 95, 99, 100]
            for p in percentiles:
                value = np.percentile(self.full_df[coord], p)
                print(f"{p}th percentile: {value:.6f}")

            print(f"Mean: {self.full_df[coord].mean():.6f}")
            print(f"Std: {self.full_df[coord].std():.6f}")
            print(f"Skewness: {self.full_df[coord].skew():.6f}")
            print(f"Kurtosis: {self.full_df[coord].kurtosis():.6f}")

        # Non-zero analysis
        print("\nAnalysis of non-zero points:")
        for coord in ['x', 'y', 'z']:
            non_zero = self.full_df[self.full_df[coord] != 0][coord]
            print(f"\n{coord.upper()} coordinate (non-zero):")
            print(f"Count: {len(non_zero)}")
            print(f"Mean: {non_zero.mean():.6f}")
            print(f"Std: {non_zero.std():.6f}")
            print(f"Min: {non_zero.min():.6f}")
            print(f"Max: {non_zero.max():.6f}")

    def analyze_points_per_frame(self):
        """Analyze and visualize points per frame distribution"""
        points_per_frame = [len(d['new_x']) for d in self.train_data]

        print("\nPoints per frame:")
        print(f"Mean: {np.mean(points_per_frame):.2f}")
        print(f"Std: {np.std(points_per_frame):.2f}")
        print(f"Min: {np.min(points_per_frame)}")
        print(f"Max: {np.max(points_per_frame)}")

        plt.figure(figsize=(10, 5))
        plt.hist(points_per_frame, bins=50)
        plt.title('Distribution of Points per Frame')
        plt.xlabel('Number of Points')
        plt.ylabel('Frequency')
        plt.show()

def main():
    # Initialize analyzer
```

```
    data_path = '/content/drive/MyDrive/action/data/processed/mmr_action/dat
    analyzer = MMRDataAnalyzer(data_path)

    # Run analyses
    print("\nAnalyzing training data...")

    # Single frame analysis
    print("\nSingle Frame Analysis:")
    frame_df = analyzer.get_frame_data(analyzer.train_data[0])
    print("\nFrame Statistics:")
    print(frame_df.describe())

    # Visualize first frame
    print("\nVisualizing first frame...")
    analyzer.visualize_frame(0)

    # Full dataset analysis
    print("\nFull Dataset Analysis:")
    full_df = analyzer.analyze_dataset()
    print("\nDataset Statistics:")
    print(full_df.describe())

    # Plot distributions
    analyzer.plot_coordinate_distributions()

    # Action analysis
    action_stats = analyzer.analyze_by_action()
    if action_stats:
        print("\nAnalysis by Action:")
        for action, stats in action_stats.items():
            print(f"\nAction {action} Statistics:")
            print(stats)

    # Points per frame analysis
    analyzer.analyze_points_per_frame()

    # Detailed statistics
    analyzer.print_detailed_statistics()

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

```
Analyzing training data...

Single Frame Analysis:

Frame Statistics:
       frame_num           x           y           z
count      110.0  110.000000  110.000000  110.000000
mean         0.0    0.047783    0.536345   -0.038962
std          0.0    0.349468    0.975685    0.212701
min          0.0   -1.158141    0.000000   -0.868396
```

```
min          0.0   -1.136141    0.000000   -0.000390
25%          0.0    0.000000    0.000000    0.000000
50%          0.0    0.000000    0.000000    0.000000
75%          0.0    0.000000    0.577146    0.000000
max          0.0    1.202685    3.890844    0.614703

Visualizing first frame...
```




```
Full Dataset Analysis:

Dataset Statistics:
          frame_num             x             y             z
count  1.873696e+07  1.873696e+07  1.873696e+07  1.873696e+07
mean   8.516750e+04  7.687180e-02  5.204076e-01 -5.673060e-02
std    4.917177e+04  3.291378e-01  9.468305e-01  2.585428e-01
min    0.000000e+00 -1.399950e+00  0.000000e+00 -1.399992e+00
25%    4.258375e+04  0.000000e+00  0.000000e+00  0.000000e+00
50%    8.516750e+04  0.000000e+00  0.000000e+00  0.000000e+00
```

Small tool functions

```
1 import numpy as np
2 from sklearn.preprocessing import MinMaxScaler
3
```

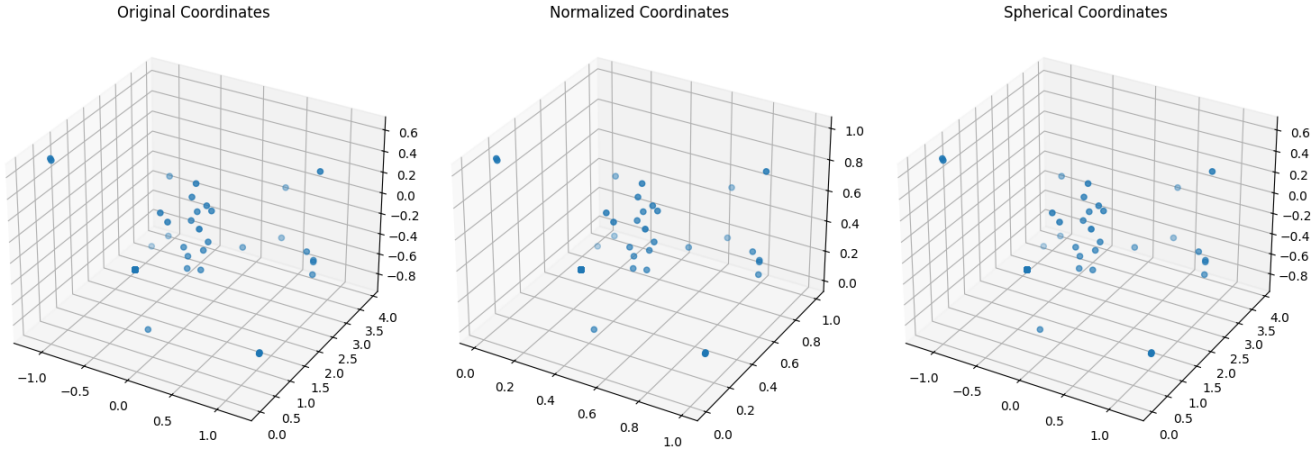
```
class MMRUtils:
    @staticmethod
    def normalize(x, x_min=None, x_max=None):
        """Normalize data to [0,1] range"""
        if x_min is None:
            x_min = np.min(x)
        if x_max is None:
            x_max = np.max(x)
        return (x - x_min) / (x_max - x_min)

    @staticmethod
    def normalize_points(points):
        """Normalize point cloud data"""
        normalized = np.zeros_like(points)
        for i in range(points.shape[1]):
            normalized[:, i] = MMRUtils.normalize(points[:, i])
        return normalized

    @staticmethod
    def weight_by_intensity(data_point):
        """Convert intensity values to weights in [0,1] range"""
        if 'intensity' in data_point:
            intensities = data_point['intensity']
            return MinMaxScaler().fit_transform(
                np.asarray(intensities).reshape(-1,1)).reshape(1,-1)[0]
        return None

    @staticmethod
    def to_vertical(array):
        """Convert array to vertical format"""
        return array.reshape(-1, 1)

    @staticmethod
    def to_horizontal(array):
        """Convert array to horizontal format"""
        return array.reshape(1, -1)

    @staticmethod
    def vector_angle(vec1, vec2):
        """Calculate normalized angle between two vectors"""
        return np.abs(np.dot(vec1, vec2)) / (np.linalg.norm(vec1) * np.linal

    @staticmethod
    def cartesian_to_spherical(points):
        """Convert cartesian coordinates to spherical
        Args:
            points: Array of shape (N, 3) containing [x, y, z] coordinates
        Returns:
            Array of shape (N, 3) containing [r, theta, phi] coordinates
        """
        x, y, z = points[:, 0], points[:, 1], points[:, 2]
```

```
        r = np.sqrt(x**2 + y**2 + z**2)
        theta = np.arctan2(y, x)
        theta[theta < 0] += 2 * np.pi
        phi = np.arctan2(np.sqrt(x**2 + y**2), z)
        return np.stack([r, theta, phi], axis=1)

    @staticmethod
    def spherical_to_cartesian(points):
        """Convert spherical coordinates to cartesian
        Args:
            points: Array of shape (N, 3) containing [r, theta, phi] coordir
        Returns:
            Array of shape (N, 3) containing [x, y, z] coordinates
        """
        r, theta, phi = points[:, 0], points[:, 1], points[:, 2]
        x = r * np.cos(theta) * np.sin(phi)
        y = r * np.sin(theta) * np.sin(phi)
        z = r * np.cos(phi)
        return np.stack([x, y, z], axis=1)

    @staticmethod
    def get_cluster_colors(labels):
        """Get colors for different clusters/labels"""
        colors = ['r', 'b', 'g', 'c', 'm', 'darkorange', 'deepskyblue',
                  'blueviolet', 'crimson', 'orangered', 'k']
        return [colors[label % len(colors)] for label in labels]

    @staticmethod
    def process_frame(data_point):
        """Process a single frame of point cloud data
        Returns normalized coordinates and spherical coordinates"""
        points = data_point['new_x']

        # Normalize cartesian coordinates
        normalized = MMRUtils.normalize_points(points)

        # Convert to spherical coordinates
        spherical = MMRUtils.cartesian_to_spherical(points)

        return {
            'original': points,
            'normalized': normalized,
            'spherical': spherical
        }

# Enhanced MMRDataAnalyzer class with utility functions
class EnhancedMMRAnalyzer(MMRDataAnalyzer):
    def __init__(self, data_path):
        super().__init__(data_path)
        self.utils = MMRUtils()

```

```
    def analyze_frame_geometry(self, frame_idx=0):
        """Analyze geometric properties of a frame"""
        data_point = self.train_data[frame_idx]
        processed = self.utils.process_frame(data_point)

        # Calculate geometric properties
        points = processed['original']
        center = np.mean(points, axis=0)
        distances = np.linalg.norm(points - center, axis=1)

        print("\nGeometric Analysis:")
        print(f"Center of mass: {center}")
        print(f"Mean distance from center: {np.mean(distances):.4f}")
        print(f"Max distance from center: {np.max(distances):.4f}")

        # Visualize in both coordinate systems
        fig = plt.figure(figsize=(15, 5))

        # Original coordinates
        ax1 = fig.add_subplot(131, projection='3d')
        ax1.scatter(points[:, 0], points[:, 1], points[:, 2])
        ax1.set_title('Original Coordinates')

        # Normalized coordinates
        ax2 = fig.add_subplot(132, projection='3d')
        norm_points = processed['normalized']
        ax2.scatter(norm_points[:, 0], norm_points[:, 1], norm_points[:, 2])
        ax2.set_title('Normalized Coordinates')

        # Spherical coordinates
        ax3 = fig.add_subplot(133, projection='3d')
        sph_points = processed['spherical']
        ax3.scatter(sph_points[:, 0] * np.sin(sph_points[:, 2]) * np.cos(sph
                    sph_points[:, 0] * np.sin(sph_points[:, 2]) * np.sin(sph_
                    sph_points[:, 0] * np.cos(sph_points[:, 2]))
        ax3.set_title('Spherical Coordinates')

        plt.tight_layout()
        plt.show()

        return processed

# Example usage
if __name__ == "__main__":
    data_path = '/content/drive/MyDrive/action/data/processed/mmr_action/dat
    analyzer = EnhancedMMRAnalyzer(data_path)

    # Analyze a frame with new utilities
    frame_data = analyzer.analyze_frame_geometry(0)

    # Example of using utilities
```

```
    points = analyzer.train_data[0]['new_x']
    normalized = analyzer.utils.normalize_points(points)
    spherical = analyzer.utils.cartesian_to_spherical(points)

    print("\nPoint Cloud Statistics:")
    print(f"Original range: [{points.min():.4f}, {points.max():.4f}]")
    print(f"Normalized range: [{normalized.min():.4f}, {normalized.max():.4f
    print(f"Spherical coordinates shape: {spherical.shape}")
```

```
Geometric Analysis:
Center of mass: [ 0.04778343  0.53634458 -0.03896176]
Mean distance from center: 0.8495
Max distance from center: 3.3704
```




```
Point Cloud Statistics:
Original range: [-1.1581, 3.8908]
Normalized range: [0.0000, 1.0000]
Spherical coordinates shape: (110, 3)
```

```
75%          0.000000       1.174127       0.000000
```

Plot data points

```
count  408430.000000  408430.000000  408430.000000
```

```
import os
import numpy as np
import pandas as pd
import pickle
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

```

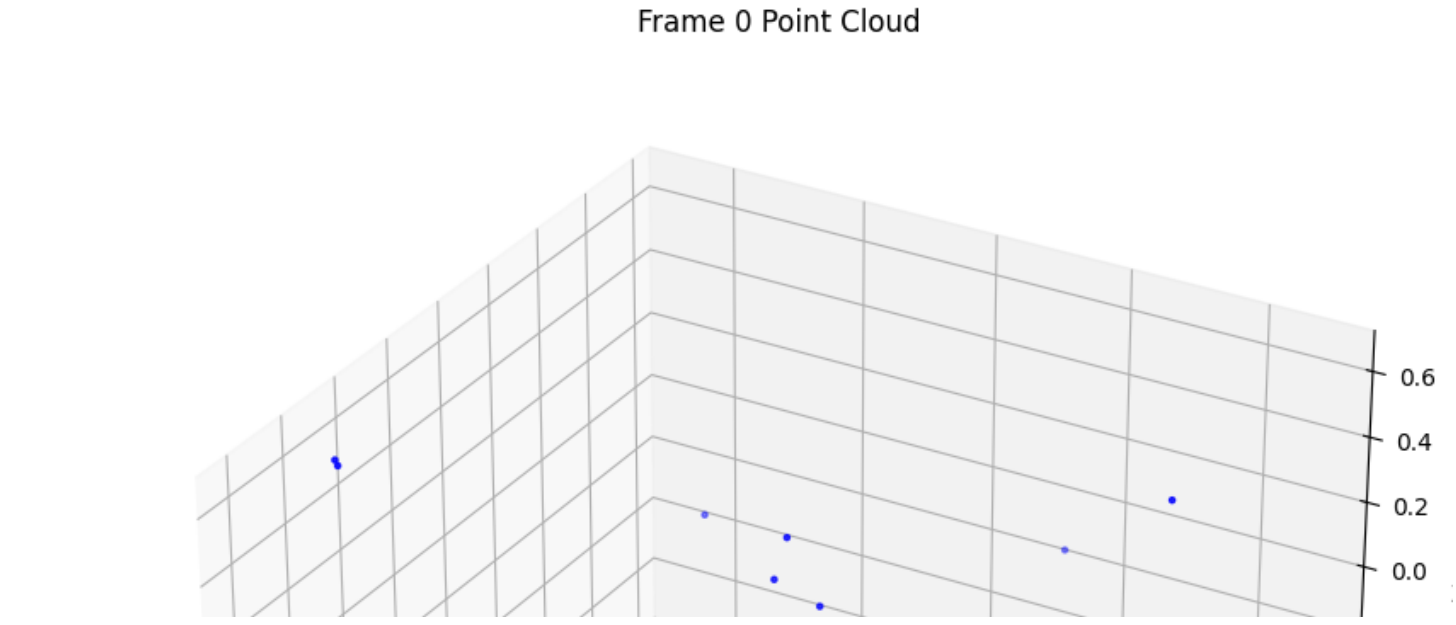
```python
class MMRUtils:
    @staticmethod
    def normalize_points(points):
        """Normalize point cloud data to [0,1] range"""
        min_vals = np.min(points, axis=0)
        max_vals = np.max(points, axis=0)
        normalized = (points - min_vals) / (max_vals - min_vals)
        return normalized

    @staticmethod
    def cartesian_to_spherical(points):
        """Convert cartesian to spherical coordinates"""
        x, y, z = points[:, 0], points[:, 1], points[:, 2]
        r = np.sqrt(x**2 + y**2 + z**2)
        theta = np.arctan2(y, x)
        phi = np.arccos(z/r)
        return np.stack([r, theta, phi], axis=1)

    @staticmethod
    def spherical_to_cartesian(points):
        """Convert spherical to cartesian coordinates"""
        r, theta, phi = points[:, 0], points[:, 1], points[:, 2]
        x = r * np.sin(phi) * np.cos(theta)
        y = r * np.sin(phi) * np.sin(theta)
        z = r * np.cos(phi)
        return np.stack([x, y, z], axis=1)

class EnhancedMMRAnalyzer:
    def __init__(self, data_path):
        self.utils = MMRUtils()
        self.data_path = data_path
        self.load_data()

    def load_data(self):
        """Load and verify data"""
        try:
            with open(self.data_path, 'rb') as f:
                self.data = pickle.load(f)
            self.train_data = self.data['train']
            print(f"Loaded {len(self.train_data)} training samples")
        except Exception as e:
            print(f"Error loading data: {e}")
            raise

    def visualize_frame(self, frame_idx=0):
        """Basic frame visualization"""
        try:
            data_point = self.train_data[frame_idx]
            points = data_point['new_x']

            # Create simple 3D scatter plot
```

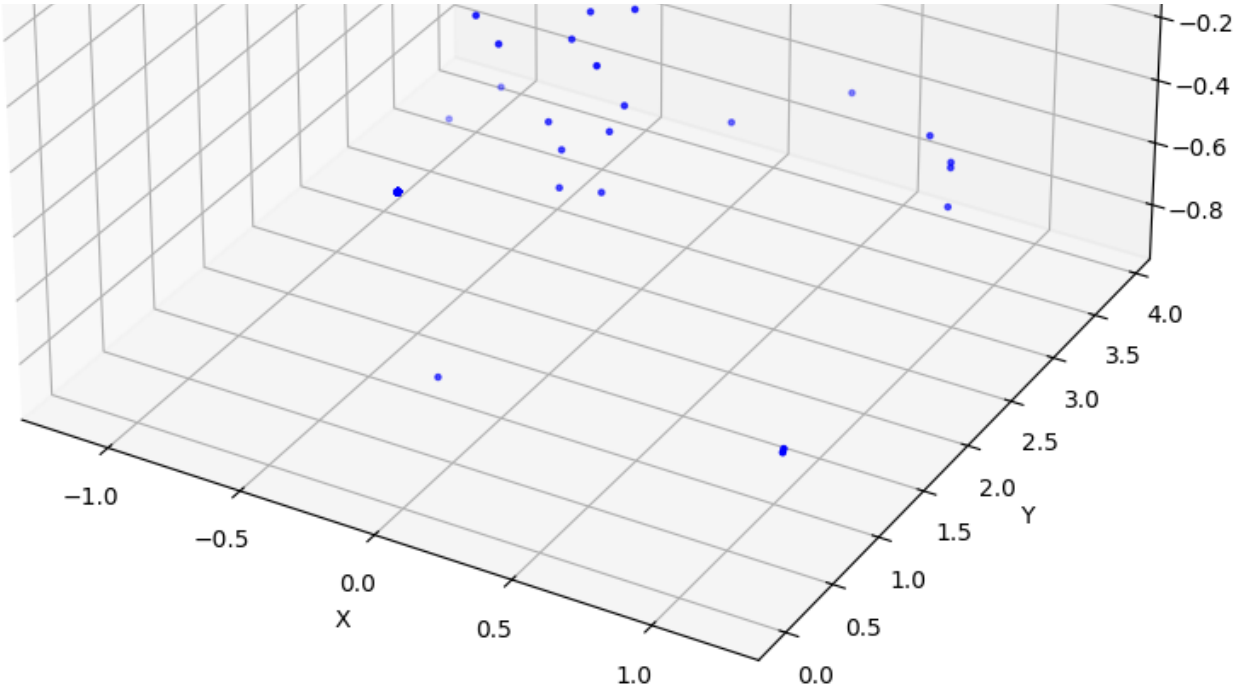
```
            fig = plt.figure(figsize=(10, 10))
            ax = fig.add_subplot(111, projection='3d')

            # Plot points
            scatter = ax.scatter(points[:, 0], points[:, 1], points[:, 2],
                                 c='blue', marker='.')

            ax.set_xlabel('X')
            ax.set_ylabel('Y')
            ax.set_zlabel('Z')
            ax.set_title(f'Frame {frame_idx} Point Cloud')

            # Print statistics
            print("\nFrame Statistics:")
            print(f"Number of points: {len(points)}")
            print(f"X range: [{points[:, 0].min():.4f}, {points[:, 0].max():
            print(f"Y range: [{points[:, 1].min():.4f}, {points[:, 1].max():
            print(f"Z range: [{points[:, 2].min():.4f}, {points[:, 2].max():

            plt.show()
            return points

        except Exception as e:
            print(f"Error visualizing frame: {e}")
            raise

    def visualize_frame_enhanced(self, frame_idx=0):
        """Enhanced visualization with multiple views"""
        try:
            data_point = self.train_data[frame_idx]
            points = data_point['new_x']

            # Calculate geometric properties
            center = np.mean(points, axis=0)
            distances = np.linalg.norm(points - center, axis=1)

            # Print analysis
            print("\nGeometric Analysis:")
            print(f"Center of mass: {center}")
            print(f"Mean distance from center: {np.mean(distances):.4f}")
            print(f"Max distance from center: {np.max(distances):.4f}")

            # Create figure with three views
            fig = plt.figure(figsize=(15, 5))

            # Original coordinates
            ax1 = fig.add_subplot(131, projection='3d')
            ax1.scatter(points[:, 0], points[:, 1], points[:, 2], c='blue',
            ax1.set_title('Original Coordinates')
            ax1.set_xlabel('X')
            ax1.set_ylabel('Y')
```

```
            ax1.set_zlabel('Z')

            # Normalized coordinates
            norm_points = self.utils.normalize_points(points)
            ax2 = fig.add_subplot(132, projection='3d')
            ax2.scatter(norm_points[:, 0], norm_points[:, 1], norm_points[:,
                        c='red', marker='.')
            ax2.set_title('Normalized Coordinates')
            ax2.set_xlabel('X')
            ax2.set_ylabel('Y')
            ax2.set_zlabel('Z')

            # Spherical coordinates view
            sph_points = self.utils.cartesian_to_spherical(points)
            sph_cart = self.utils.spherical_to_cartesian(sph_points)
            ax3 = fig.add_subplot(133, projection='3d')
            ax3.scatter(sph_cart[:, 0], sph_cart[:, 1], sph_cart[:, 2],
                        c='green', marker='.')
            ax3.set_title('Spherical Coordinates')
            ax3.set_xlabel('X')
            ax3.set_ylabel('Y')
            ax3.set_zlabel('Z')

            plt.tight_layout()
            plt.show()

            # Print ranges
            print("\nPoint Cloud Statistics:")
            print(f"Original range: [{points.min():.4f}, {points.max():.4f}]
            print(f"Normalized range: [{norm_points.min():.4f}, {norm_points
            print(f"Spherical coordinates shape: {sph_points.shape}")

            return points

        except Exception as e:
            print(f"Error in enhanced visualization: {e}")
            raise

    def compare_frames(self, frame_indices):
        """Compare multiple frames"""
        try:
            n_frames = len(frame_indices)
            fig = plt.figure(figsize=(5*n_frames, 5))

            for i, idx in enumerate(frame_indices):
                ax = fig.add_subplot(1, n_frames, i+1, projection='3d')
                points = self.train_data[idx]['new_x']

                ax.scatter(points[:, 0], points[:, 1], points[:, 2],
                           c='blue', marker='.')
                ax.set_title(f'Frame {idx}')
```

```
                ax.set_xlabel('X')
                ax.set_ylabel('Y')
                ax.set_zlabel('Z')

            plt.tight_layout()
            plt.show()

        except Exception as e:
            print(f"Error comparing frames: {e}")
            raise

# Example usage
if __name__ == "__main__":
    data_path = '/content/drive/MyDrive/action/data/processed/mmr_action/dat
    analyzer = EnhancedMMRAnalyzer(data_path)

    # Basic visualization
    print("\nBasic Visualization:")
    analyzer.visualize_frame(0)

    # Enhanced visualization
    print("\nEnhanced Visualization:")
    analyzer.visualize_frame_enhanced(0)

    # Compare multiple frames
    print("\nComparing Multiple Frames:")
    analyzer.compare_frames([0, 1, 2])
```

```
Loaded 170336 training samples

Basic Visualization:

Frame Statistics:
Number of points: 110
X range: [-1.1581, 1.2027]
Y range: [0.0000, 3.8908]
Z range: [-0.8684, 0.6147]
```






```
Enhanced Visualization:

Geometric Analysis:
Center of mass: [ 0.04778343  0.53634458 -0.03896176]
Mean distance from center: 0.8495
Max distance from center: 3.3704
<ipython-input-9-f8f139a9df25>:23: RuntimeWarning: invalid value encountere
  phi = np.arccos(z/r)
```

Choose frame window

```
WINDOW = 5
```



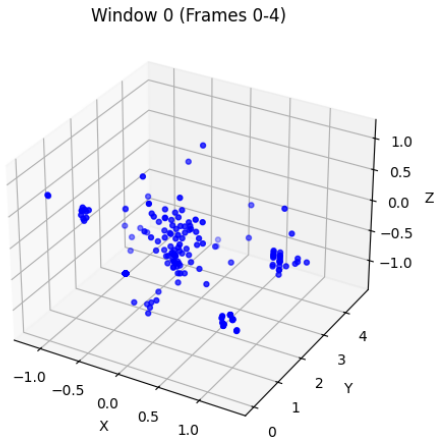
```
def analyze_mmr_windows(data_path, window_size=5):
    """Analyze MMR data with sliding windows"""

    # Load data
    with open(data_path, 'rb') as f:
        all_data = pickle.load(f)
    train_data = all_data['train']

    print(f"Analyzing {len(train_data)} frames with window size {window_size

    # Single frame analysis
    frame_points = []
    for data_point in train_data:
        points = data_point['new_x']
```

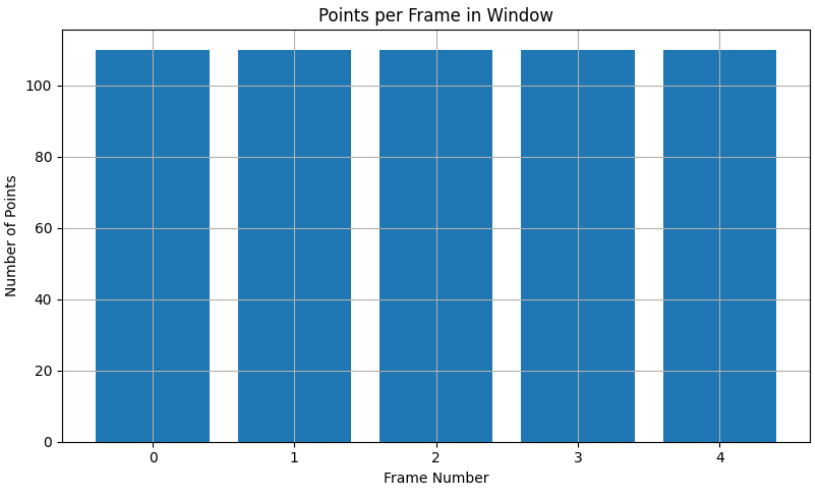
```
        frame_points.append(len(points))

    frame_series = pd.Series(frame_points)
    print("\nPoints per Frame Statistics:")
    print(frame_series.describe())

    # Window analysis
    window_points = []
    num_windows = len(train_data) // window_size

    for i in range(num_windows):
        start_idx = i * window_size
        end_idx = start_idx + window_size
        window_count = sum(frame_points[start_idx:end_idx])
        window_points.append(window_count)

    window_series = pd.Series(window_points)
    print(f"\nPoints per {window_size}-Frame Window Statistics:")
    print(window_series.describe())

    # Visualize first window
    visualize_window(train_data, 0, window_size)

    # Compare first few windows
    compare_windows(train_data, [0, 1, 2], window_size)

    return frame_series, window_series

def visualize_window(train_data, window_idx, window_size):
    """Visualize all points in a window"""
    start_idx = window_idx * window_size
    end_idx = start_idx + window_size

    # Collect points
    all_points = []
    frame_counts = []

    for i in range(start_idx, min(end_idx, len(train_data))):
        points = train_data[i]['new_x']
        all_points.append(points)
        frame_counts.append(len(points))

    all_points = np.vstack(all_points)

    # Create visualization
    fig = plt.figure(figsize=(15, 5))

    # 3D scatter plot
    ax1 = fig.add_subplot(121, projection='3d')
    scatter = ax1.scatter(all_points[:, 0], all_points[:, 1], all_points[:,
                          c='blue', marker='.', s=50)
```

```
    ax1.set_xlabel('X')
    ax1.set_ylabel('Y')
    ax1.set_zlabel('Z')
    ax1.set_title(f'Window {window_idx} (Frames {start_idx}-{end_idx-1})')
    ax1.grid(True)

    # Point count distribution
    ax2 = fig.add_subplot(122)
    ax2.bar(range(start_idx, end_idx), frame_counts)
    ax2.set_xlabel('Frame Number')
    ax2.set_ylabel('Number of Points')
    ax2.set_title('Points per Frame in Window')
    ax2.grid(True)

    plt.tight_layout()
    display(plt.gcf())
    plt.close()

    # Print statistics
    print(f"\nWindow {window_idx} Statistics:")
    print(f"Total frames: {len(frame_counts)}")
    print(f"Total points: {len(all_points)}")
    print(f"Average points per frame: {np.mean(frame_counts):.2f}")
    print(f"Point range: [{all_points.min():.4f}, {all_points.max():.4f}]")

    return all_points, frame_counts

def compare_windows(train_data, window_indices, window_size):
    """Compare multiple windows"""
    n_windows = len(window_indices)
    fig = plt.figure(figsize=(5*n_windows, 5))

    for i, idx in enumerate(window_indices):
        start_idx = idx * window_size
        end_idx = start_idx + window_size

        # Collect window points
        all_points = []
        for j in range(start_idx, min(end_idx, len(train_data))):
            points = train_data[j]['new_x']
            all_points.append(points)
        all_points = np.vstack(all_points)

        # Plot
        ax = fig.add_subplot(1, n_windows, i+1, projection='3d')
        ax.scatter(all_points[:, 0], all_points[:, 1], all_points[:, 2],
                   c='blue', marker='.', s=50)
        ax.set_title(f'Window {idx}\n(Frames {start_idx}-{end_idx-1})')
        ax.set_xlabel('X')
        ax.set_ylabel('Y')
        ax.set_zlabel('Z')
```

```
        ax.grid(True)

    plt.tight_layout()
    display(plt.gcf())
    plt.close()

# Example usage
if __name__ == "__main__":
    # Set parameters
    data_path = '/content/drive/MyDrive/action/data/processed/mmr_action/dat
    WINDOW_SIZE = 5

    # Run analysis
    frame_stats, window_stats = analyze_mmr_windows(data_path, WINDOW_SIZE)
```

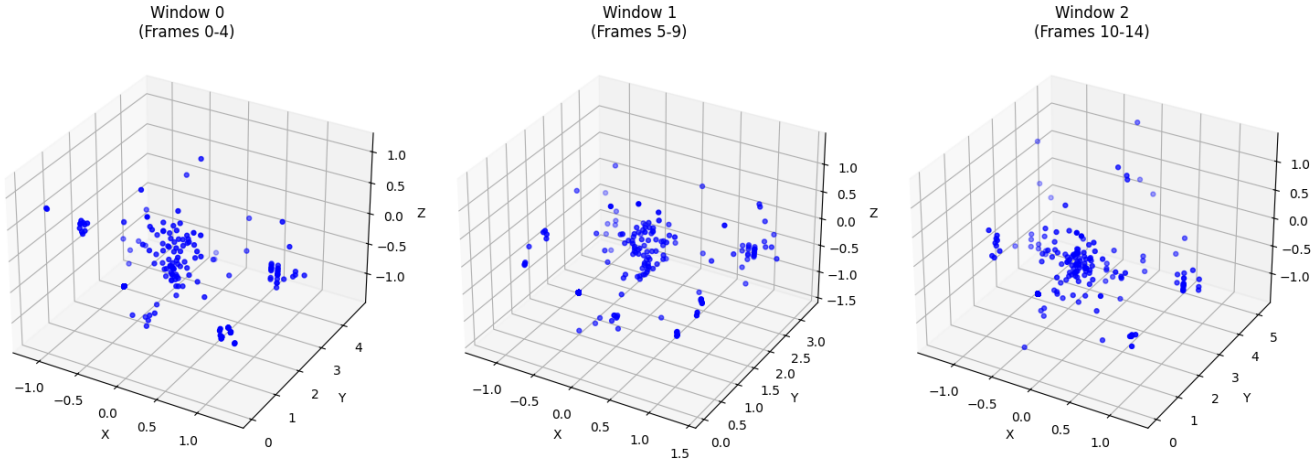
```
Analyzing 170336 frames with window size 5

Points per Frame Statistics:
count    170336.0
mean         110.0
std            0.0
min          110.0
25%          110.0
50%          110.0
75%          110.0
max          110.0
dtype: float64

Points per 5-Frame Window Statistics:
count    34067.0
mean       550.0
std          0.0
min        550.0
25%        550.0
50%        550.0
75%        550.0
max        550.0
dtype: float64
```






```
Window 0 Statistics:
Total frames: 5
Total points: 550
```

```
Average points per frame: 110.00
Point range: [-1.3165, 4.7211]
```



DBSCAN clustering

```
from sklearn.cluster import DBSCAN
```

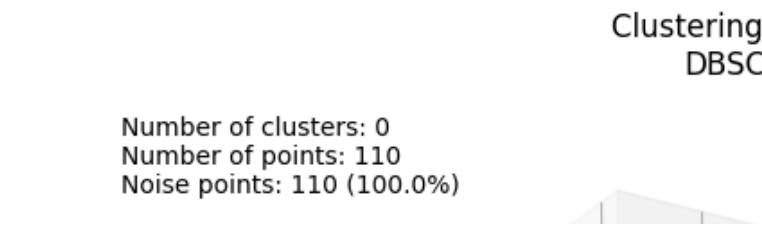
```
# Define global variables
DBSCAN_EPS = 0.3
DBSCAN_SAMPLES = 6
```

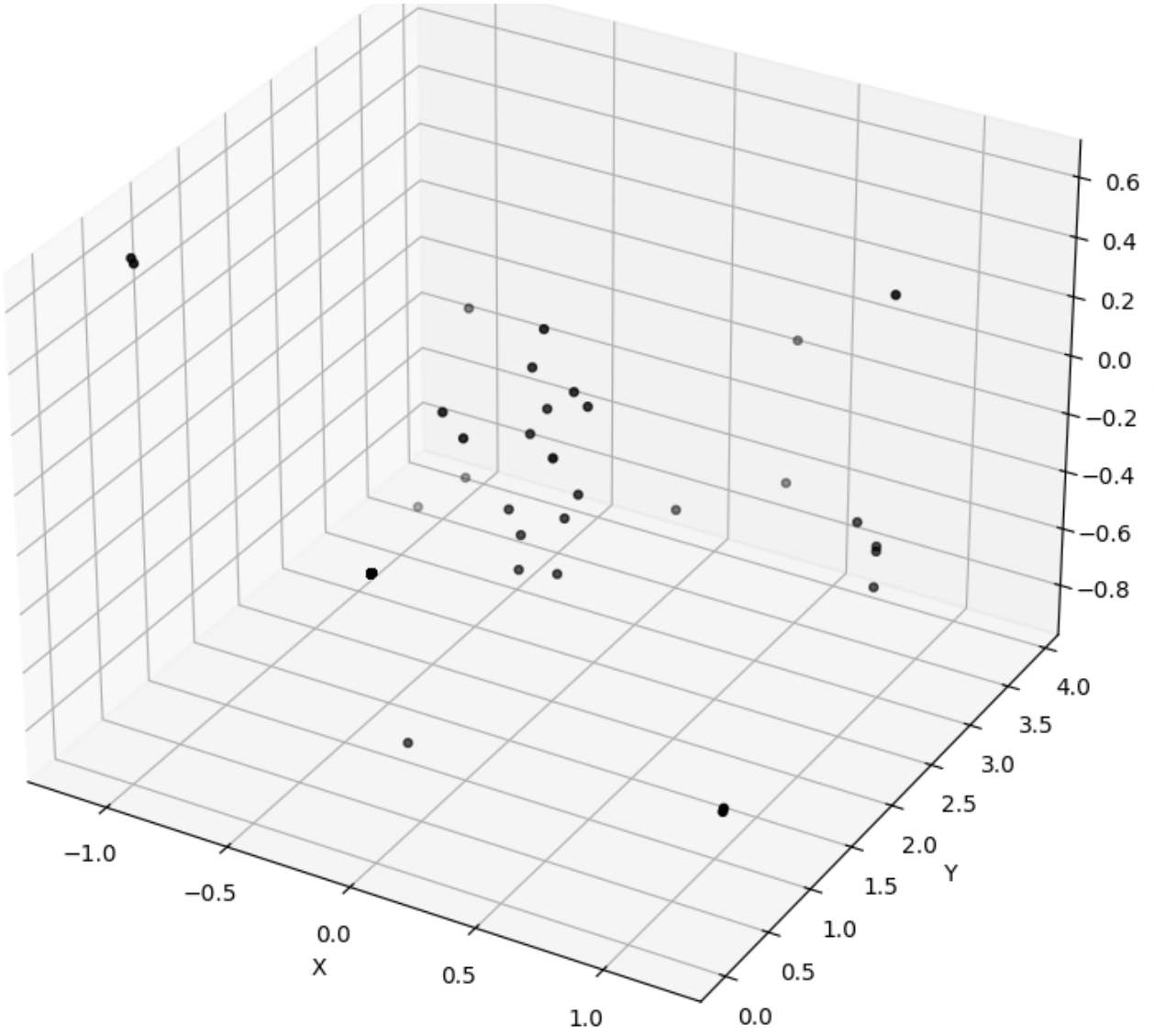
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler

def get_colors(label_list):
    """Get colors for cluster labels"""
    color_list = ['r', 'b', 'g', 'c', 'm', 'darkorange', 'deepskyblue',
                  'blueviolet', 'crimson', 'orangered', 'k']
    return [color_list[label % len(color_list)] for label in label_list]

def normalize_weights(points):
    """Normalize point weights to [0,1]"""
    return MinMaxScaler().fit_transform(
        np.linalg.norm(points, axis=1).reshape(-1, 1)
    ).ravel()

def model_cluster(points, model, use_weights=True, show_plot=True, return_cl
    """
    Cluster point cloud data using the provided model

    Args:
        points: numpy array of shape (N, 3) containing point cloud coordinat
```

```
        model: clustering model (e.g., DBSCAN, KMeans)
        use_weights: whether to use point distances as weights
        show_plot: whether to show clustering visualization
        return_cluster: whether to return the clustering model

    Returns:
        clustering model if return_cluster is True
    """
    # Prepare weights if needed
    if use_weights:
        sample_weights = normalize_weights(points)
    else:
        sample_weights = None

    # Perform clustering
    clustering = model.fit(points, sample_weight=sample_weights)

    # Visualize if requested
    if show_plot:
        fig = plt.figure(figsize=(12, 8))
        ax = fig.add_subplot(111, projection='3d')

        # Plot points colored by cluster
        scatter = ax.scatter(points[:, 0], points[:, 1], points[:, 2],
                             c=get_colors(clustering.labels_),
                             marker='.',
                             s=50)

        # Add cluster centers if available (e.g., for KMeans)
        if hasattr(model, 'cluster_centers_'):
            centers = model.cluster_centers_
            ax.scatter(centers[:, 0], centers[:, 1], centers[:, 2],
                       c='black',
                       marker='*',
                       s=200,
                       label='Cluster Centers')

        # Configure plot
        ax.set_xlabel('X')
        ax.set_ylabel('Y')
        ax.set_zlabel('Z')
        ax.set_title(f'Clustering Results\n{model.__class__.__name__}')
        ax.grid(True)

        # Add statistics annotation
        n_clusters = len(set(clustering.labels_)) - (1 if -1 in clustering.l
        stats_text = f'Number of clusters: {n_clusters}\n'
        stats_text += f'Number of points: {len(points)}\n'
        if -1 in clustering.labels_:
            n_noise = np.sum(clustering.labels_ == -1)
            stats_text += f'Noise points: {n_noise} ({n_noise/len(points)*10
```

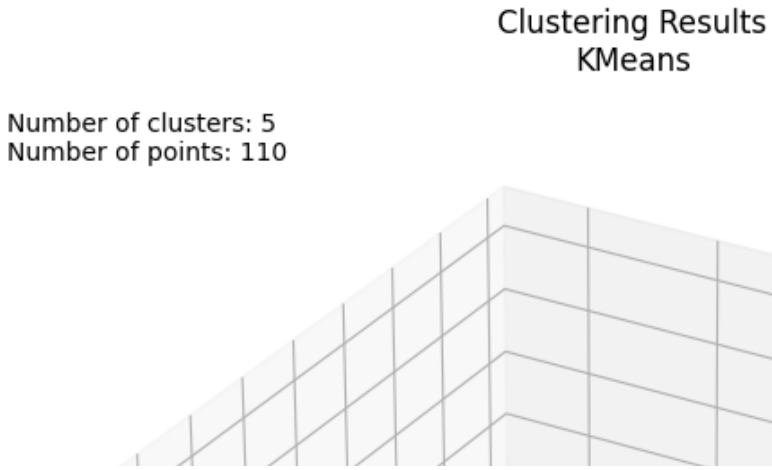
```
        ax.text2D(0.02, 0.98, stats_text,
                  transform=ax.transAxes,
                  verticalalignment='top')

        plt.tight_layout()
        display(plt.gcf())
        plt.close()

        # Print cluster sizes
        print("\nCluster Sizes:")
        unique_labels = sorted(set(clustering.labels_))
        for label in unique_labels:
            count = np.sum(clustering.labels_ == label)
            if label == -1:
                print(f"Noise points: {count} ({count/len(points)*100:.1f}%)
            else:
                print(f"Cluster {label}: {count} points ({count/len(points)*

    if return_cluster:
        return clustering

# Example usage
if __name__ == "__main__":
    # Load your data
    data_path = '/content/drive/MyDrive/action/data/processed/mmr_action/dat
    with open(data_path, 'rb') as f:
        data = pickle.load(f)

    # Get points from first frame
    points = data['train'][0]['new_x']

    # Try different clustering models
    from sklearn.cluster import DBSCAN, KMeans

    # DBSCAN clustering
    dbscan = DBSCAN(eps=0.3, min_samples=5)
    print("\nDBSCAN Clustering:")
    dbscan_result = model_cluster(points, dbscan, use_weights=True, return_c

    # KMeans clustering
    kmeans = KMeans(n_clusters=5, random_state=42)
    print("\nKMeans Clustering:")
    kmeans_result = model_cluster(points, kmeans, use_weights=True, return_c
```

```
DBSCAN Clustering:
```

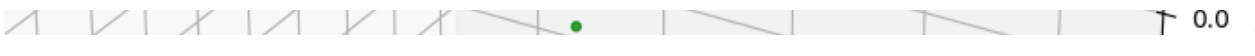





```
Cluster Sizes:
Noise points: 110 (100.0%)

KMeans Clustering:
```



Compare with and without sample_weight



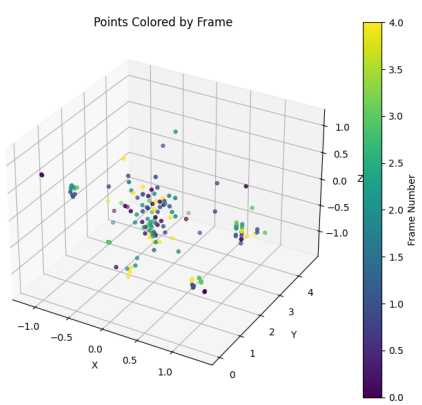

```
1 def cluster_window(data_points, start_frame, window_size, eps=0.3, min_sampl
```

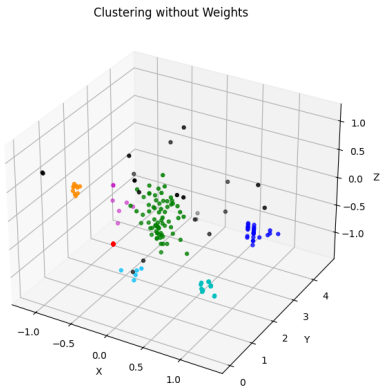
```
    """
    Cluster points in a specific frame window

    Args:
        data_points: List of frame data points
        start_frame: Starting frame index
        window_size: Number of frames in window
        eps: DBSCAN epsilon parameter
        min_samples: DBSCAN min_samples parameter
        use_weights: Whether to use intensity weights
    """
    # Collect points from window
    window_points = []
    frame_indices = []

    end_frame = start_frame + window_size
    for i in range(start_frame, end_frame):
        if i < len(data_points):
            points = data_points[i]['new_x']
            window_points.append(points)
            frame_indices.extend([i] * len(points))

    window_points = np.vstack(window_points)
    frame_indices = np.array(frame_indices)

    # Create DBSCAN model
    dbscan = DBSCAN(eps=eps, min_samples=min_samples)

    # Prepare sample weights if needed
    if use_weights:
        weights = normalize_weights(window_points)
    else:
        weights = None

    # Perform clustering
    clustering = dbscan.fit(window_points, sample_weight=weights)

    # Visualize results
    fig = plt.figure(figsize=(20, 6))

    # Original points colored by frame
    ax1 = fig.add_subplot(131, projection='3d')
    scatter1 = ax1.scatter(window_points[:, 0], window_points[:, 1], window_
                           c=frame_indices, cmap='viridis',
                           marker='.', s=50)
    ax1.set_title('Points Colored by Frame')
    ax1.set_xlabel('X')
    ax1.set_ylabel('Y')
    ax1.set_zlabel('Z')
    plt.colorbar(scatter1, label='Frame Number')

```

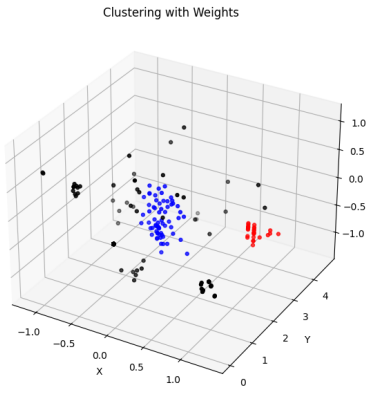
```
    # Clustering without weights
    dbscan_no_weights = DBSCAN(eps=eps, min_samples=min_samples)
    labels_no_weights = dbscan_no_weights.fit_predict(window_points)

    ax2 = fig.add_subplot(132, projection='3d')
    scatter2 = ax2.scatter(window_points[:, 0], window_points[:, 1], window_
                           c=get_colors(labels_no_weights),
                           marker='.', s=50)
    ax2.set_title('Clustering without Weights')
    ax2.set_xlabel('X')
    ax2.set_ylabel('Y')
    ax2.set_zlabel('Z')

    # Clustering with weights
    ax3 = fig.add_subplot(133, projection='3d')
    scatter3 = ax3.scatter(window_points[:, 0], window_points[:, 1], window_
                           c=get_colors(clustering.labels_),
                           marker='.', s=50)
    ax3.set_title('Clustering with Weights')
    ax3.set_xlabel('X')
    ax3.set_ylabel('Y')
    ax3.set_zlabel('Z')

    plt.tight_layout()
    display(plt.gcf())
    plt.close()

    # Print statistics
    print(f"\nWindow Statistics (Frames {start_frame}-{end_frame-1}):")
    print(f"Total points: {len(window_points)}")

    print("\nClustering without weights:")
    n_clusters_no_weights = len(set(labels_no_weights)) - (1 if -1 in labels
    print(f"Number of clusters: {n_clusters_no_weights}")
    if -1 in labels_no_weights:
        n_noise = np.sum(labels_no_weights == -1)
        print(f"Noise points: {n_noise} ({n_noise/len(window_points)*100:.1f

    print("\nClustering with weights:")
    n_clusters = len(set(clustering.labels_)) - (1 if -1 in clustering.label
    print(f"Number of clusters: {n_clusters}")
    if -1 in clustering.labels_:
        n_noise = np.sum(clustering.labels_ == -1)
        print(f"Noise points: {n_noise} ({n_noise/len(window_points)*100:.1f

    return clustering, window_points, frame_indices

def analyze_window_clusters(clustering, window_points, frame_indices):
    """Analyze clusters in a window"""
    unique_labels = sorted(set(clustering.labels_))

```

```
    # Create visualization
    n_clusters = len(unique_labels)
    fig_cols = min(3, n_clusters)
    fig_rows = (n_clusters + fig_cols - 1) // fig_cols

    fig = plt.figure(figsize=(6*fig_cols, 5*fig_rows))

    for i, label in enumerate(unique_labels):
        mask = clustering.labels_ == label
        cluster_points = window_points[mask]
        cluster_frames = frame_indices[mask]

        ax = fig.add_subplot(fig_rows, fig_cols, i+1, projection='3d')
        scatter = ax.scatter(cluster_points[:, 0],
                             cluster_points[:, 1],
                             cluster_points[:, 2],
                             c=cluster_frames,
                             cmap='viridis',
                             marker='.',
                             s=50)

        title = "Noise Points" if label == -1 else f"Cluster {label}"
        title += f"\n{len(cluster_points)} points"
        ax.set_title(title)
        ax.set_xlabel('X')
        ax.set_ylabel('Y')
        ax.set_zlabel('Z')
        plt.colorbar(scatter, label='Frame Number')

    plt.tight_layout()
    display(plt.gcf())
    plt.close()

# Example usage
if __name__ == "__main__":
    # Parameters
    WINDOW = 5
    DBSCAN_EPS = 0.3
    DBSCAN_SAMPLES = 5
    START_FRAME = 0

    # Load data
    data_path = '/content/drive/MyDrive/action/data/processed/mmr_action/dat
    with open(data_path, 'rb') as f:
        data = pickle.load(f)

    # Perform clustering
    clustering, points, frames = cluster_window(
        data['train'],
        start_frame=START_FRAME,
        window_size=WINDOW,
```

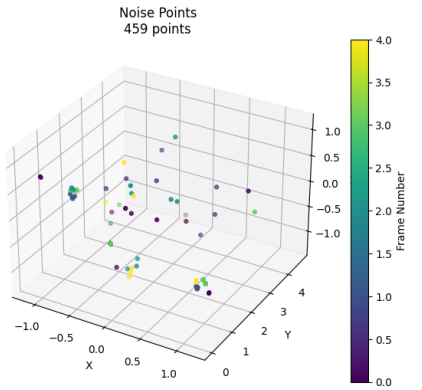
```
155         eps=DBSCAN_EPS,
156         min_samples=DBSCAN_SAMPLES
157     )
158 
159     # Analyze clusters
160     analyze_window_clusters(clustering, points, frames)
```

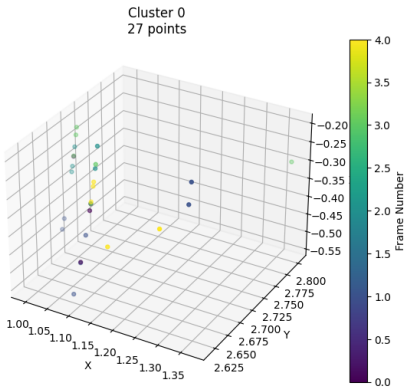


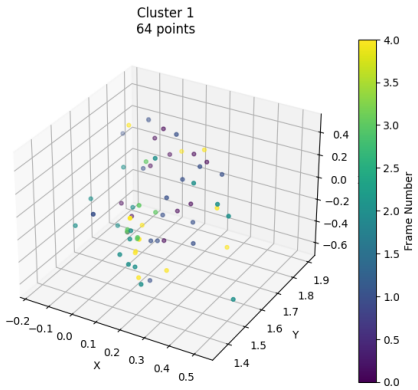





```
Window Statistics (Frames 0-4):
Total points: 550

Clustering without weights:
Number of clusters: 7
Noise points: 20 (3.6%)

Clustering with weights:
Number of clusters: 2
Noise points: 459 (83.5%)
```







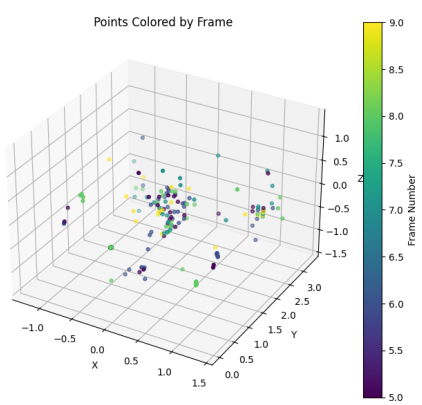

```
def cluster_window(data_points, start_frame, window_size, eps=0.3, min_sampl
    """
    Cluster points in a specific frame window

    Args:
```

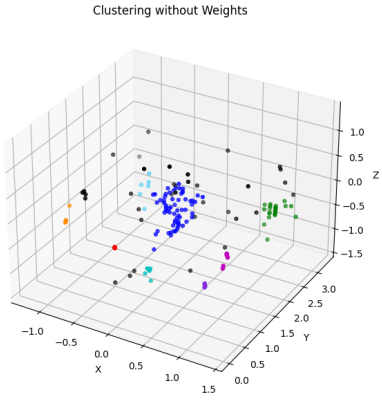
```
        data_points: List of frame data points
        start_frame: Starting frame index
        window_size: Number of frames in window
        eps: DBSCAN epsilon parameter
        min_samples: DBSCAN min_samples parameter
        use_weights: Whether to use intensity weights
    """
    # Collect points from window
    window_points = []
    frame_indices = []

    end_frame = start_frame + window_size
    for i in range(start_frame, end_frame):
        if i < len(data_points):
            points = data_points[i]['new_x']
            window_points.append(points)
            frame_indices.extend([i] * len(points))

    window_points = np.vstack(window_points)
    frame_indices = np.array(frame_indices)

    # Create DBSCAN model
    dbscan = DBSCAN(eps=eps, min_samples=min_samples)

    # Prepare sample weights if needed
    if use_weights:
        weights = normalize_weights(window_points)
    else:
        weights = None

    # Perform clustering
    clustering = dbscan.fit(window_points, sample_weight=weights)

    # Visualize results
    fig = plt.figure(figsize=(20, 6))

    # Original points colored by frame
    ax1 = fig.add_subplot(131, projection='3d')
    scatter1 = ax1.scatter(window_points[:, 0], window_points[:, 1], window_
                           c=frame_indices, cmap='viridis',
                           marker='.', s=50)
    ax1.set_title('Points Colored by Frame')
    ax1.set_xlabel('X')
    ax1.set_ylabel('Y')
    ax1.set_zlabel('Z')
    plt.colorbar(scatter1, label='Frame Number')

    # Clustering without weights
    dbscan_no_weights = DBSCAN(eps=eps, min_samples=min_samples)
    labels_no_weights = dbscan_no_weights.fit_predict(window_points)

```

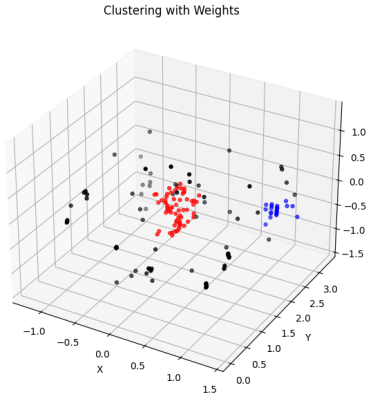
```
    ax2 = fig.add_subplot(132, projection='3d')
    scatter2 = ax2.scatter(window_points[:, 0], window_points[:, 1], window_
                           c=get_colors(labels_no_weights),
                           marker='.', s=50)
    ax2.set_title('Clustering without Weights')
    ax2.set_xlabel('X')
    ax2.set_ylabel('Y')
    ax2.set_zlabel('Z')

    # Clustering with weights
    ax3 = fig.add_subplot(133, projection='3d')
    scatter3 = ax3.scatter(window_points[:, 0], window_points[:, 1], window_
                           c=get_colors(clustering.labels_),
                           marker='.', s=50)
    ax3.set_title('Clustering with Weights')
    ax3.set_xlabel('X')
    ax3.set_ylabel('Y')
    ax3.set_zlabel('Z')

    plt.tight_layout()
    display(plt.gcf())
    plt.close()

    # Print statistics
    print(f"\nWindow Statistics (Frames {start_frame}-{end_frame-1}):")
    print(f"Total points: {len(window_points)}")

    print("\nClustering without weights:")
    n_clusters_no_weights = len(set(labels_no_weights)) - (1 if -1 in labels
    print(f"Number of clusters: {n_clusters_no_weights}")
    if -1 in labels_no_weights:
        n_noise = np.sum(labels_no_weights == -1)
        print(f"Noise points: {n_noise} ({n_noise/len(window_points)*100:.1f

    print("\nClustering with weights:")
    n_clusters = len(set(clustering.labels_)) - (1 if -1 in clustering.label
    print(f"Number of clusters: {n_clusters}")
    if -1 in clustering.labels_:
        n_noise = np.sum(clustering.labels_ == -1)
        print(f"Noise points: {n_noise} ({n_noise/len(window_points)*100:.1f

    return clustering, window_points, frame_indices

def analyze_window_clusters(clustering, window_points, frame_indices):
    """Analyze clusters in a window"""
    unique_labels = sorted(set(clustering.labels_))

    # Create visualization
    n_clusters = len(unique_labels)
    fig_cols = min(3, n_clusters)
    fig_rows = (n_clusters + fig_cols - 1) // fig_cols
```

```
108
109     fig = plt.figure(figsize=(6*fig_cols, 5*fig_rows))
110
111     for i, label in enumerate(unique_labels):
112         mask = clustering.labels_ == label
113         cluster_points = window_points[mask]
114         cluster_frames = frame_indices[mask]
115
116         ax = fig.add_subplot(fig_rows, fig_cols, i+1, projection='3d')
117         scatter = ax.scatter(cluster_points[:, 0],
118                              cluster_points[:, 1],
119                              cluster_points[:, 2],
120                              c=cluster_frames,
121                              cmap='viridis',
122                              marker='.',
123                              s=50)
124
125         title = "Noise Points" if label == -1 else f"Cluster {label}"
126         title += f"\n{len(cluster_points)} points"
127         ax.set_title(title)
128         ax.set_xlabel('X')
129         ax.set_ylabel('Y')
130         ax.set_zlabel('Z')
131         plt.colorbar(scatter, label='Frame Number')
132
133     plt.tight_layout()
134     display(plt.gcf())
135     plt.close()
136
137 # Example usage
138 if __name__ == "__main__":
139     # Parameters
140     WINDOW = 5
141     DBSCAN_EPS = 0.3
142     DBSCAN_SAMPLES = 5
143     START_FRAME = 5
144
145     # Load data
146     data_path = '/content/drive/MyDrive/action/data/processed/mmr_action/dat
147     with open(data_path, 'rb') as f:
148         data = pickle.load(f)
149
150     # Perform clustering
151     clustering, points, frames = cluster_window(
152         data['train'],
153         start_frame=START_FRAME,
154         window_size=WINDOW,
155         eps=DBSCAN_EPS,
156         min_samples=DBSCAN_SAMPLES
157     )
158
```

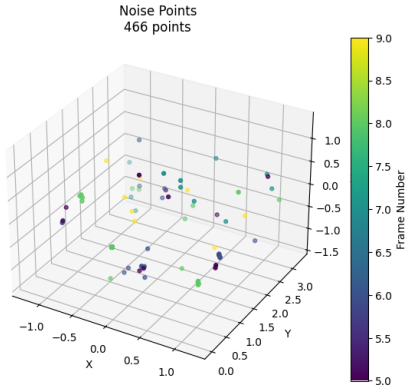
```
159     # Analyze clusters
160     analyze_window_clusters(clustering, points, frames)
```

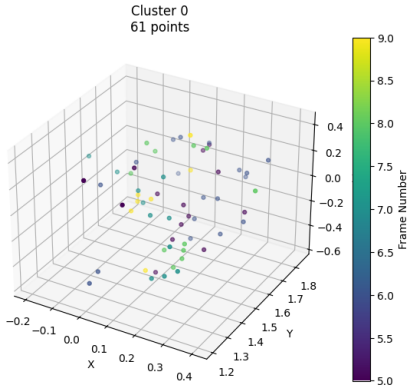


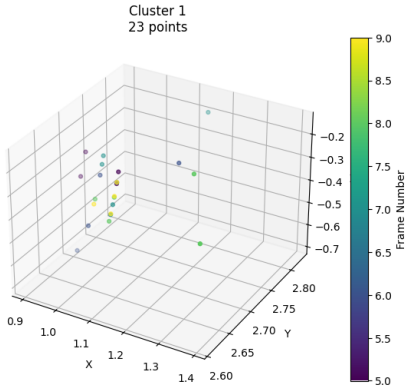





```
Window Statistics (Frames 5-9):
Total points: 550

Clustering without weights:
Number of clusters: 8
Noise points: 35 (6.4%)

Clustering with weights:
Number of clusters: 2
Noise points: 466 (84.7%)
```







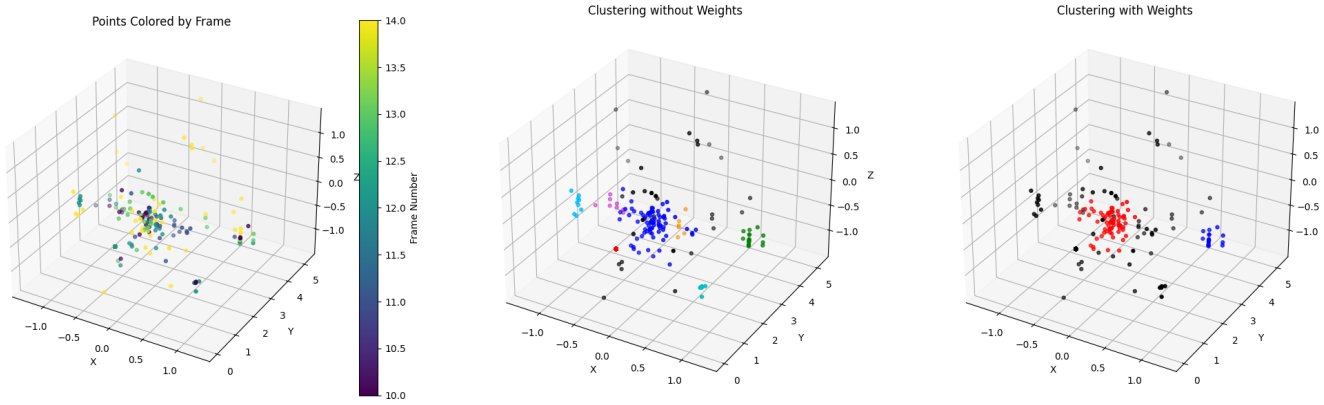

```
1 def cluster_window(data_points, start_frame, window_size, eps=0.3, min_sampl
2     """
```

```
    Cluster points in a specific frame window

    Args:
        data_points: List of frame data points
        start_frame: Starting frame index
        window_size: Number of frames in window
        eps: DBSCAN epsilon parameter
        min_samples: DBSCAN min_samples parameter
        use_weights: Whether to use intensity weights
    """
    # Collect points from window
    window_points = []
    frame_indices = []

    end_frame = start_frame + window_size
    for i in range(start_frame, end_frame):
        if i < len(data_points):
            points = data_points[i]['new_x']
            window_points.append(points)
            frame_indices.extend([i] * len(points))

    window_points = np.vstack(window_points)
    frame_indices = np.array(frame_indices)

    # Create DBSCAN model
    dbscan = DBSCAN(eps=eps, min_samples=min_samples)

    # Prepare sample weights if needed
    if use_weights:
        weights = normalize_weights(window_points)
    else:
        weights = None

    # Perform clustering
    clustering = dbscan.fit(window_points, sample_weight=weights)

    # Visualize results
    fig = plt.figure(figsize=(20, 6))

    # Original points colored by frame
    ax1 = fig.add_subplot(131, projection='3d')
    scatter1 = ax1.scatter(window_points[:, 0], window_points[:, 1], window_
                           c=frame_indices, cmap='viridis',
                           marker='.', s=50)
    ax1.set_title('Points Colored by Frame')
    ax1.set_xlabel('X')
    ax1.set_ylabel('Y')
    ax1.set_zlabel('Z')
    plt.colorbar(scatter1, label='Frame Number')

    # Clustering without weights
```

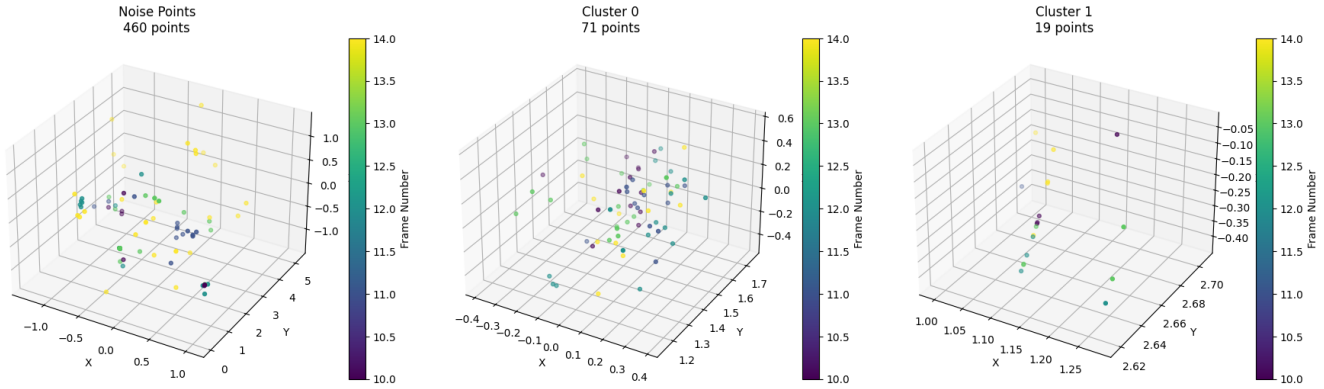
```
    dbscan_no_weights = DBSCAN(eps=eps, min_samples=min_samples)
    labels_no_weights = dbscan_no_weights.fit_predict(window_points)

    ax2 = fig.add_subplot(132, projection='3d')
    scatter2 = ax2.scatter(window_points[:, 0], window_points[:, 1], window_
                           c=get_colors(labels_no_weights),
                           marker='.', s=50)
    ax2.set_title('Clustering without Weights')
    ax2.set_xlabel('X')
    ax2.set_ylabel('Y')
    ax2.set_zlabel('Z')

    # Clustering with weights
    ax3 = fig.add_subplot(133, projection='3d')
    scatter3 = ax3.scatter(window_points[:, 0], window_points[:, 1], window_
                           c=get_colors(clustering.labels_),
                           marker='.', s=50)
    ax3.set_title('Clustering with Weights')
    ax3.set_xlabel('X')
    ax3.set_ylabel('Y')
    ax3.set_zlabel('Z')

    plt.tight_layout()
    display(plt.gcf())
    plt.close()

    # Print statistics
    print(f"\nWindow Statistics (Frames {start_frame}-{end_frame-1}):")
    print(f"Total points: {len(window_points)}")

    print("\nClustering without weights:")
    n_clusters_no_weights = len(set(labels_no_weights)) - (1 if -1 in labels
    print(f"Number of clusters: {n_clusters_no_weights}")
    if -1 in labels_no_weights:
        n_noise = np.sum(labels_no_weights == -1)
        print(f"Noise points: {n_noise} ({n_noise/len(window_points)*100:.1f

    print("\nClustering with weights:")
    n_clusters = len(set(clustering.labels_)) - (1 if -1 in clustering.label
    print(f"Number of clusters: {n_clusters}")
    if -1 in clustering.labels_:
        n_noise = np.sum(clustering.labels_ == -1)
        print(f"Noise points: {n_noise} ({n_noise/len(window_points)*100:.1f

    return clustering, window_points, frame_indices

def analyze_window_clusters(clustering, window_points, frame_indices):
    """Analyze clusters in a window"""
    unique_labels = sorted(set(clustering.labels_))

    # Create visualization
```

```
    n_clusters = len(unique_labels)
    fig_cols = min(3, n_clusters)
    fig_rows = (n_clusters + fig_cols - 1) // fig_cols

    fig = plt.figure(figsize=(6*fig_cols, 5*fig_rows))

    for i, label in enumerate(unique_labels):
        mask = clustering.labels_ == label
        cluster_points = window_points[mask]
        cluster_frames = frame_indices[mask]

        ax = fig.add_subplot(fig_rows, fig_cols, i+1, projection='3d')
        scatter = ax.scatter(cluster_points[:, 0],
                             cluster_points[:, 1],
                             cluster_points[:, 2],
                             c=cluster_frames,
                             cmap='viridis',
                             marker='.',
                             s=50)

        title = "Noise Points" if label == -1 else f"Cluster {label}"
        title += f"\n{len(cluster_points)} points"
        ax.set_title(title)
        ax.set_xlabel('X')
        ax.set_ylabel('Y')
        ax.set_zlabel('Z')
        plt.colorbar(scatter, label='Frame Number')

    plt.tight_layout()
    display(plt.gcf())
    plt.close()

# Example usage
if __name__ == "__main__":
    # Parameters
    WINDOW = 5
    DBSCAN_EPS = 0.3
    DBSCAN_SAMPLES = 5
    START_FRAME = 10

    # Load data
    data_path = '/content/drive/MyDrive/action/data/processed/mmr_action/dat
    with open(data_path, 'rb') as f:
        data = pickle.load(f)

    # Perform clustering
    clustering, points, frames = cluster_window(
        data['train'],
        start_frame=START_FRAME,
        window_size=WINDOW,
        eps=DBSCAN_EPS,
```

```
        min_samples=DBSCAN_SAMPLES
    )

    # Analyze clusters
    analyze_window_clusters(clustering, points, frames)
```




```
Window Statistics (Frames 10-14):
Total points: 550

Clustering without weights:
Number of clusters: 7
Noise points: 31 (5.6%)

Clustering with weights:
Number of clusters: 2
Noise points: 460 (83.6%)
```

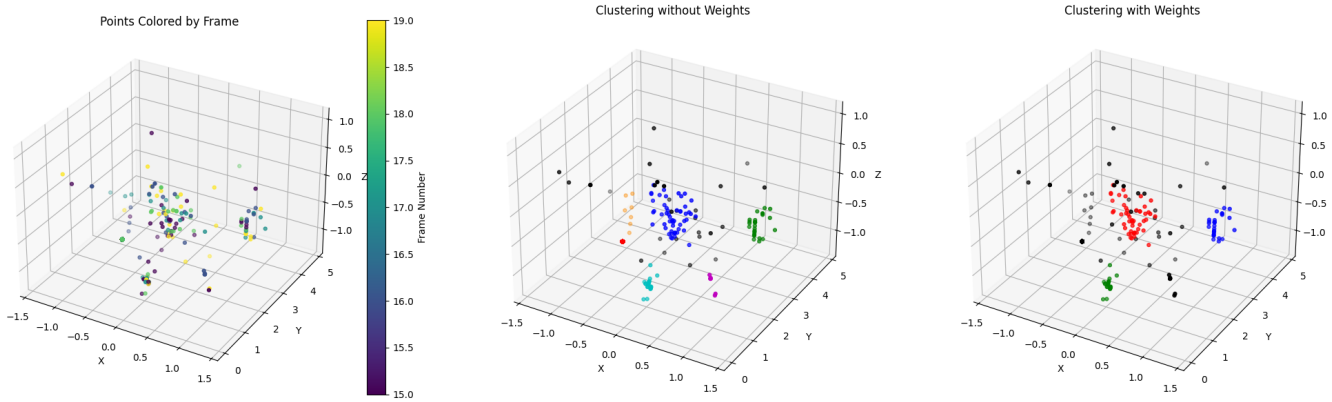


```python
def cluster_window(data_points, start_frame, window_size, eps=0.3, min_sampl
    """
    Cluster points in a specific frame window

    Args:
        data_points: List of frame data points
        start_frame: Starting frame index
        window_size: Number of frames in window
        eps: DBSCAN epsilon parameter
        min_samples: DBSCAN min_samples parameter
        use_weights: Whether to use intensity weights
    """
    # Collect points from window
    window_points = []
    frame_indices = []

    end_frame = start_frame + window_size
    for i in range(start_frame, end_frame):
        if i < len(data_points):
            points = data_points[i]['new_x']
            window_points.append(points)
            frame_indices.extend([i] * len(points))

    window_points = np.vstack(window_points)
    frame_indices = np.array(frame_indices)

    # Create DBSCAN model
    dbscan = DBSCAN(eps=eps, min_samples=min_samples)

    # Prepare sample weights if needed
    if use_weights:
        weights = normalize_weights(window_points)
    else:
        weights = None

    # Perform clustering
    clustering = dbscan.fit(window_points, sample_weight=weights)

    # Visualize results
    fig = plt.figure(figsize=(20, 6))

    # Original points colored by frame
    ax1 = fig.add_subplot(131, projection='3d')
    scatter1 = ax1.scatter(window_points[:, 0], window_points[:, 1], window_
                           c=frame_indices, cmap='viridis',
                           marker='.', s=50)
    ax1.set_title('Points Colored by Frame')
    ax1.set_xlabel('X')
    ax1.set_ylabel('Y')
    ax1.set_zlabel('Z')
    plt.colorbar(scatter1, label='Frame Number')
```

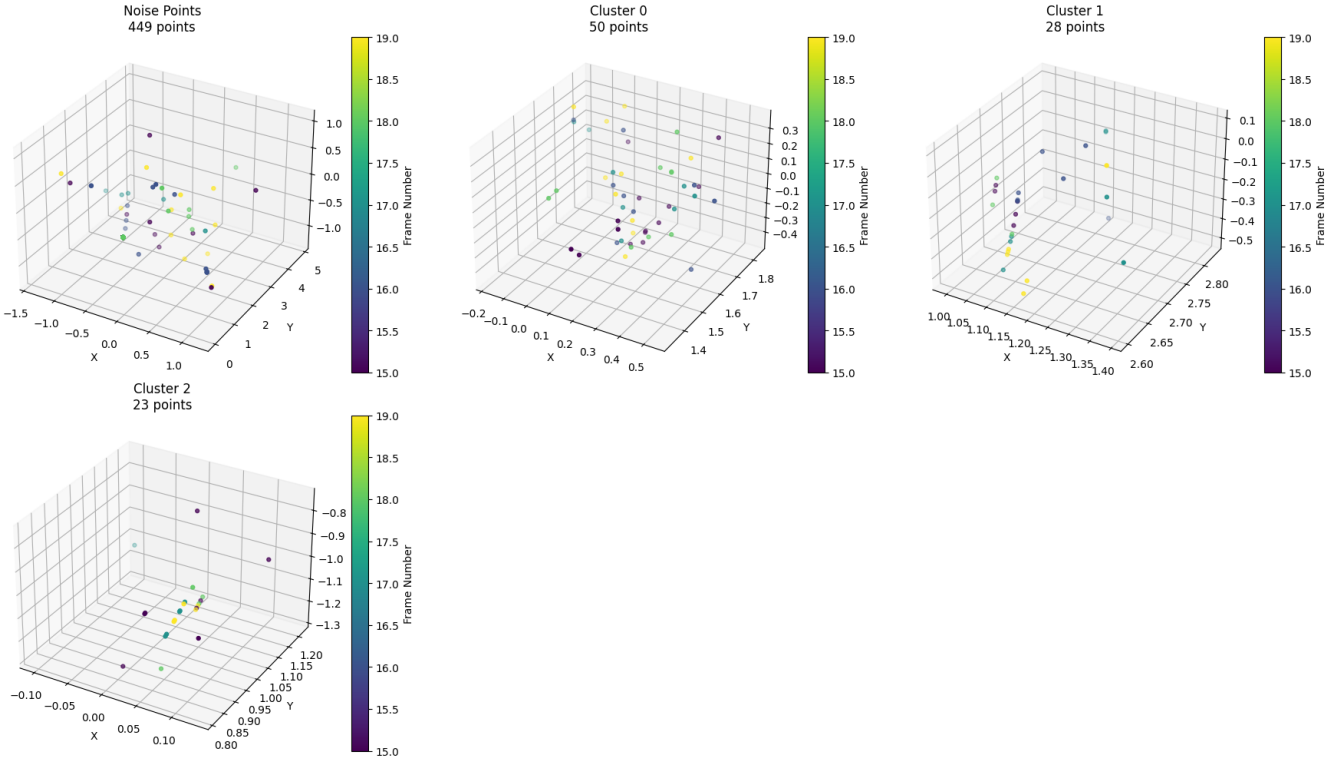
```
52
53      # Clustering without weights
54      dbscan_no_weights = DBSCAN(eps=eps, min_samples=min_samples)
55      labels_no_weights = dbscan_no_weights.fit_predict(window_points)
56
57      ax2 = fig.add_subplot(132, projection='3d')
58      scatter2 = ax2.scatter(window_points[:, 0], window_points[:, 1], window_
59                             c=get_colors(labels_no_weights),
60                             marker='.', s=50)
61      ax2.set_title('Clustering without Weights')
62      ax2.set_xlabel('X')
63      ax2.set_ylabel('Y')
64      ax2.set_zlabel('Z')
65
66      # Clustering with weights
67      ax3 = fig.add_subplot(133, projection='3d')
68      scatter3 = ax3.scatter(window_points[:, 0], window_points[:, 1], window_
69                             c=get_colors(clustering.labels_),
70                             marker='.', s=50)
71      ax3.set_title('Clustering with Weights')
72      ax3.set_xlabel('X')
73      ax3.set_ylabel('Y')
74      ax3.set_zlabel('Z')
75
76      plt.tight_layout()
77      display(plt.gcf())
78      plt.close()
79
80      # Print statistics
81      print(f"\nWindow Statistics (Frames {start_frame}-{end_frame-1}):")
82      print(f"Total points: {len(window_points)}")
83
84      print("\nClustering without weights:")
85      n_clusters_no_weights = len(set(labels_no_weights)) - (1 if -1 in labels
86      print(f"Number of clusters: {n_clusters_no_weights}")
87      if -1 in labels_no_weights:
88          n_noise = np.sum(labels_no_weights == -1)
89          print(f"Noise points: {n_noise} ({n_noise/len(window_points)*100:.1f
90
91      print("\nClustering with weights:")
92      n_clusters = len(set(clustering.labels_)) - (1 if -1 in clustering.label
93      print(f"Number of clusters: {n_clusters}")
94      if -1 in clustering.labels_:
95          n_noise = np.sum(clustering.labels_ == -1)
96          print(f"Noise points: {n_noise} ({n_noise/len(window_points)*100:.1f
97
98      return clustering, window_points, frame_indices
99
100 def analyze_window_clusters(clustering, window_points, frame_indices):
101     """Analyze clusters in a window"""
102     unique_labels = sorted(set(clustering.labels_))
```

```
103
104     # Create visualization
105     n_clusters = len(unique_labels)
106     fig_cols = min(3, n_clusters)
107     fig_rows = (n_clusters + fig_cols - 1) // fig_cols
108
109     fig = plt.figure(figsize=(6*fig_cols, 5*fig_rows))
110
111     for i, label in enumerate(unique_labels):
112         mask = clustering.labels_ == label
113         cluster_points = window_points[mask]
114         cluster_frames = frame_indices[mask]
115
116         ax = fig.add_subplot(fig_rows, fig_cols, i+1, projection='3d')
117         scatter = ax.scatter(cluster_points[:, 0],
118                              cluster_points[:, 1],
119                              cluster_points[:, 2],
120                              c=cluster_frames,
121                              cmap='viridis',
122                              marker='.',
123                              s=50)
124
125         title = "Noise Points" if label == -1 else f"Cluster {label}"
126         title += f"\n{len(cluster_points)} points"
127         ax.set_title(title)
128         ax.set_xlabel('X')
129         ax.set_ylabel('Y')
130         ax.set_zlabel('Z')
131         plt.colorbar(scatter, label='Frame Number')
132
133     plt.tight_layout()
134     display(plt.gcf())
135     plt.close()
136
137 # Example usage
138 if __name__ == "__main__":
139     # Parameters
140     WINDOW = 5
141     DBSCAN_EPS = 0.3
142     DBSCAN_SAMPLES = 5
143     START_FRAME = 15
144
145     # Load data
146     data_path = '/content/drive/MyDrive/action/data/processed/mmr_action/dat
147     with open(data_path, 'rb') as f:
148         data = pickle.load(f)
149
150     # Perform clustering
151     clustering, points, frames = cluster_window(
152         data['train'],
153         start_frame=START_FRAME,
```

```
        window_size=WINDOW,
        eps=DBSCAN_EPS,
        min_samples=DBSCAN_SAMPLES
    )

    # Analyze clusters
    analyze_window_clusters(clustering, points, frames)
```




```
Window Statistics (Frames 15-19):
Total points: 550

Clustering without weights:
Number of clusters: 6
Noise points: 28 (5.1%)

Clustering with weights:
Number of clusters: 3
Noise points: 449 (81.6%)
```



Change DBSCAN eps, min_samples [TODO]

Estimate boundary for each cluster (extension, orientation)

```
def ModelCluster(data_points, model, sample_weight=True, show_plot=True, ret
    """
    Cluster point cloud data
    Args:
        data_points: numpy array of points
        model: clustering model (e.g., DBSCAN)
        sample_weight: whether to use weights
        show_plot: whether to show visualization
        return_cluster: whether to return clustering object
    """
    if sample_weight:
        weights = MinMaxScaler().fit_transform(
            np.linalg.norm(data_points, axis=1).reshape(-1,1)
        ).ravel()
    else:
        weights = None

    clustering = model.fit(data_points, sample_weight=weights)

    if show_plot:
        fig = plt.figure()
        ax = fig.gca(projection='3d')
        PlotSetting(ax)

        scatter = ax.scatter(data_points[:, 0],
                             data_points[:, 1],
                             data_points[:, 2],
                             c=get_colors(clustering.labels_),
                             marker='.')
        plt.show()

```

```
    if return_cluster:
        return clustering

def analyze_window(data, start_frame, window_size=5):
    """Analyze a window of frames"""
    # Collect points from window frames
    window_points = []
    for i in range(start_frame, start_frame + window_size):
        if i < len(data):
            points = data[i]['new_x']
            window_points.append(points)

    # Stack all points
    window_points = np.vstack(window_points)

    return window_points

# Example usage
if __name__ == "__main__":
    # Load data
    with open('/content/drive/MyDrive/action/data/processed/mmr_action/data.
        data = pickle.load(f)
    train_data = data['train']

    # Set parameters
    WINDOW = 5
    DBSCAN_EPS = 0.3
    DBSCAN_SAMPLES = 6

    # Process windows
    for startFrame in range(0, 20, WINDOW):
        print(f"\nAnalyzing frames {startFrame} to {startFrame+WINDOW-1}")

        # Get window points
        window_points = analyze_window(train_data, startFrame, WINDOW)

        # Create and apply DBSCAN
        dbscan = DBSCAN(eps=DBSCAN_EPS, min_samples=DBSCAN_SAMPLES)
        clustering = ModelCluster(window_points, dbscan,
                                  sample_weight=True,
                                  show_plot=False,
                                  return_cluster=True)

        # Analyze clusters
        print("Cluster Analysis:")
        ClusterAnalysis(clustering.labels_)
```

```
Analyzing frames 0 to 4
Cluster Analysis:
--------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call
last)
<ipython-input-27-34b188df72e7> in <cell line: 50>()
     75         # Analyze clusters
     76         print("Cluster Analysis:")
---> 77         ClusterAnalysis(clustering.labels_)

NameError: name 'ClusterAnalysis' is not defined
```

```
# Parameters
WINDOW = 5
startFrame = 0
DBSCAN_EPS = 0.3
DBSCAN_SAMPLES = 6

# Load data
# Load data
with open('/content/drive/MyDrive/action/data/processed/mmr_action/data.pkl'
        data = pickle.load(f)
train_data = data['train']

def Weight(datalist):
    """Normalize intensity values to [0,1] for weights"""
    return MinMaxScaler().fit_transform(
        np.asarray(datalist['intensity']).reshape(-1,1)
    ).reshape(1,-1)[0]

def ClusterCenter(points, weights):
    """Calculate weighted centroid position of cluster"""
    return np.average(points, axis=0, weights=weights)

# Get points for the window
window_points = []
for i in range(startFrame, startFrame + WINDOW):
    if i < len(train_data):
        points = train_data[i]['new_x']
        window_points.append(points)
window_points = np.vstack(window_points)

# Perform clustering
dbscan = DBSCAN(eps=DBSCAN_EPS, min_samples=DBSCAN_SAMPLES)
clustering = ModelCluster(window_points, dbscan,
                          sample_weight=True,
                          show_plot=False,
                          return_cluster=True)

# Get points for cluster 0
cluster_mask = clustering.labels_ == 0
cluster_points = window_points[cluster_mask]

# Calculate cluster center
weights = normalize_weights(cluster_points)  # Using the normalize function
cluster_center = ClusterCenter(cluster_points, weights)
print("Cluster 0 center:", cluster_center)
```

```
Cluster 0 center: [ 1.33793459  2.73593783 -0.27574565]
```

```
import numpy as np
import pandas as pd
```

```
3
4 def debug_clustering(clusterlist):
5     # Print basic stats
6     print(f"Number of points: {len(clusterlist)}")
7     print("\nPoint statistics:")
8     print(clusterlist.describe())
9
10    # Check weights
11    points = np.array(clusterlist[['x','y','z']])
12    weights = np.linalg.norm(points, axis=1)
13    print("\nWeight statistics:")
14    print(f"Min weight: {weights.min()}")
15    print(f"Max weight: {weights.max()}")
16    print(f"Mean weight: {weights.mean()}")
17
18    # Check cluster center
19    center = ClusterCenter(clusterlist)
20    print("\nCluster center:", center)
21
22    # Check covariance calculation
23    weights = Weight(clusterlist)
24    cov = CalcCovariance(clusterlist, center, weights)
25    print("\nCovariance matrix:")
26    print(cov)
27
28 # Add this before calculating extensions:
29 debug_clustering(clusterlist)
```

```
Number of points: 395

Point statistics:
            x      y      z
count  395.0  395.0  395.0
mean     0.0    0.0    0.0
std      0.0    0.0    0.0
min      0.0    0.0    0.0
25%      0.0    0.0    0.0
50%      0.0    0.0    0.0
75%      0.0    0.0    0.0
max      0.0    0.0    0.0

Weight statistics:
Min weight: 0.0
Max weight: 0.0
Mean weight: 0.0

Cluster center: [0. 0. 0.]

Covariance matrix:
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

```python
# After loading data
print("Raw point statistics:")
for i in range(5):
    points = train_data[i]['new_x']
    print(f"\nFrame {i}:")
    print(f"Number of points: {len(points)}")
    print(f"Min values: {np.min(points, axis=0)}")
    print(f"Max values: {np.max(points, axis=0)}")

# Before DBSCAN
print("\nWindow points before clustering:")
print(f"Shape: {window_points.shape}")
print(f"Non-zero points: {np.count_nonzero(window_points)}")
print(f"Sample of points:\n{window_points[:5]}")
```

```
Raw point statistics:

Frame 0:
Number of points: 110
Min values: [-1.15814066  0.         -0.86839569]
Max values: [1.20268464 3.89084435 0.61470288]

Frame 1:
Number of points: 110
Min values: [-0.9290579  0.        -1.2256074]
Max values: [1.30068111 4.72105169 0.69817191]

Frame 2:
Number of points: 110
Min values: [-0.83615208  0.         -1.01736093]
Max values: [1.08432519 2.75244379 1.11984921]

Frame 3:
Number of points: 110
Min values: [-0.74324632  0.         -0.5372054 ]
Max values: [1.37195122 2.82800937 0.56259519]

Frame 4:
Number of points: 110
Min values: [-0.91633111  0.         -1.31651485]
Max values: [1.28286362 4.65428591 0.98492074]

Window points before clustering:
Shape: (550, 3)
Non-zero points: 456
Sample of points:
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

```python
import numpy as np
import pandas as pd
```

```
from sklearn.preprocessing import MinMaxScaler
from scipy.optimize import linear_sum_assignment
from sklearn.cluster import DBSCAN

def HorToVer(array):
    return array.reshape(-1,1)

def VerToHor(array):
    return array.reshape(1,-1)

def VectorAngle(vec1, vec2):
    return np.abs(np.dot(vec1, vec2))/(np.linalg.norm(vec1)*np.linalg.norm(ve

def Weight(clusterlist):
    points = np.array(clusterlist[['x','y','z']])
    weights = np.linalg.norm(points, axis=1)
    weights = weights + 1e-8
    return weights / np.sum(weights)

def ClusterCenter(clusterlist):
    weights = Weight(clusterlist)
    if np.all(weights == 0):
        return np.mean(clusterlist[['x','y','z']], axis=0)
    return np.average(clusterlist[['x','y','z']], axis=0, weights=weights)

def CalcCovariance(clusterlist, cluster_center, cluster_pt_weight, orientatio
    if orientation:
        cluster_cov = np.zeros((3,3))
        for i in range(len(clusterlist.index)):
            pt_diff = np.array(clusterlist[['x','y','z']])[i]-cluster_center
            cluster_cov += cluster_pt_weight[i]*np.dot(HorToVer(pt_diff), Ver
    else:
        cluster_cov = np.zeros(3)
        for i in range(len(clusterlist.index)):
            cluster_cov[0] += cluster_pt_weight[i]*np.dot(np.array(clusterlis
                                                          np.array(clusterlist.
            cluster_cov[1] += cluster_pt_weight[i]*np.dot(np.array(clusterlis
                                                          np.array(clusterlist.
            cluster_cov[2] += cluster_pt_weight[i]*np.dot(np.array(clusterlis
                                                          np.array(clusterlist.
    cluster_cov = cluster_cov/np.sum(cluster_pt_weight)
    return cluster_cov

def ShiftAxis(cluster_extension, cluster_orientation):
    shift_index = np.zeros((3,3))
    for axis in range(3):
        for idx in range(3):
            shift_index[axis, idx] = VectorAngle(cluster_orientation[idx], np
    shift_index = linear_sum_assignment(shift_index, maximize=True)[1]

    cluster_orientation = cluster_orientation[shift_index]
    cluster_extension = cluster_extension[shift_index]
```

```
    cluster_extension = cluster_extension[shift_index]

    for idx in range(3):
        if np.dot(cluster_orientation[idx], np.identity(3)[idx]) < 0:
            cluster_orientation[idx] = -cluster_orientation[idx]
    return cluster_extension, cluster_orientation

def ClusterExtension(clusterlist, orientation=True):
    cluster_pt_weight = Weight(clusterlist)
    cluster_center = ClusterCenter(clusterlist)

    if orientation:
        cluster_cov = CalcCovariance(clusterlist, cluster_center, cluster_pt_
        cluster_extension, cluster_orientation = np.linalg.eig(cluster_cov)
        cluster_extension = np.sqrt(cluster_extension)*6
        cluster_orientation = cluster_orientation.T
        cluster_extension, cluster_orientation = ShiftAxis(cluster_extension,
    else:
        cluster_cov = CalcCovariance(clusterlist, cluster_center, cluster_pt_
        cluster_extension = np.sqrt(cluster_cov)*6
        cluster_orientation = np.identity(3)
    return cluster_extension, cluster_orientation

# Load data and run analysis
with open('/content/drive/MyDrive/action/data/processed/mmr_action/data.pkl',
    data = pickle.load(f)
train_data = data['train']

# Get window points
window_points = []
for i in range(5):  # 5 frames window
    points = train_data[i]['new_x']
    window_points.append(points)
window_points = np.vstack(window_points)

# Perform clustering
dbscan = DBSCAN(eps=0.3, min_samples=6)
clustering = dbscan.fit(window_points)

# Check for valid clusters
cluster_mask = clustering.labels_ == 0
if np.any(cluster_mask):
    clusterlist = pd.DataFrame(window_points[cluster_mask], columns=['x', 'y'

    # With orientation
    cluster_extension, cluster_orientation = ClusterExtension(clusterlist)
    print('With orientation estimation:')
    print('length:', cluster_extension[0])
    print('width:', cluster_extension[1])
    print('height:', cluster_extension[2])
    print('orientation:', cluster_orientation[np.argmax(cluster_extension)])

```

```
    # Without orientation
    cluster_extension, cluster_orientation = ClusterExtension(clusterlist, or
    print('\nWithout orientation estimation:')
    print('length:', cluster_extension[0])
    print('width:', cluster_extension[1])
    print('height:', cluster_extension[2])
    print('orientation:', cluster_orientation[np.argmax(cluster_extension)])
else:
    print("No points in cluster 0")
```

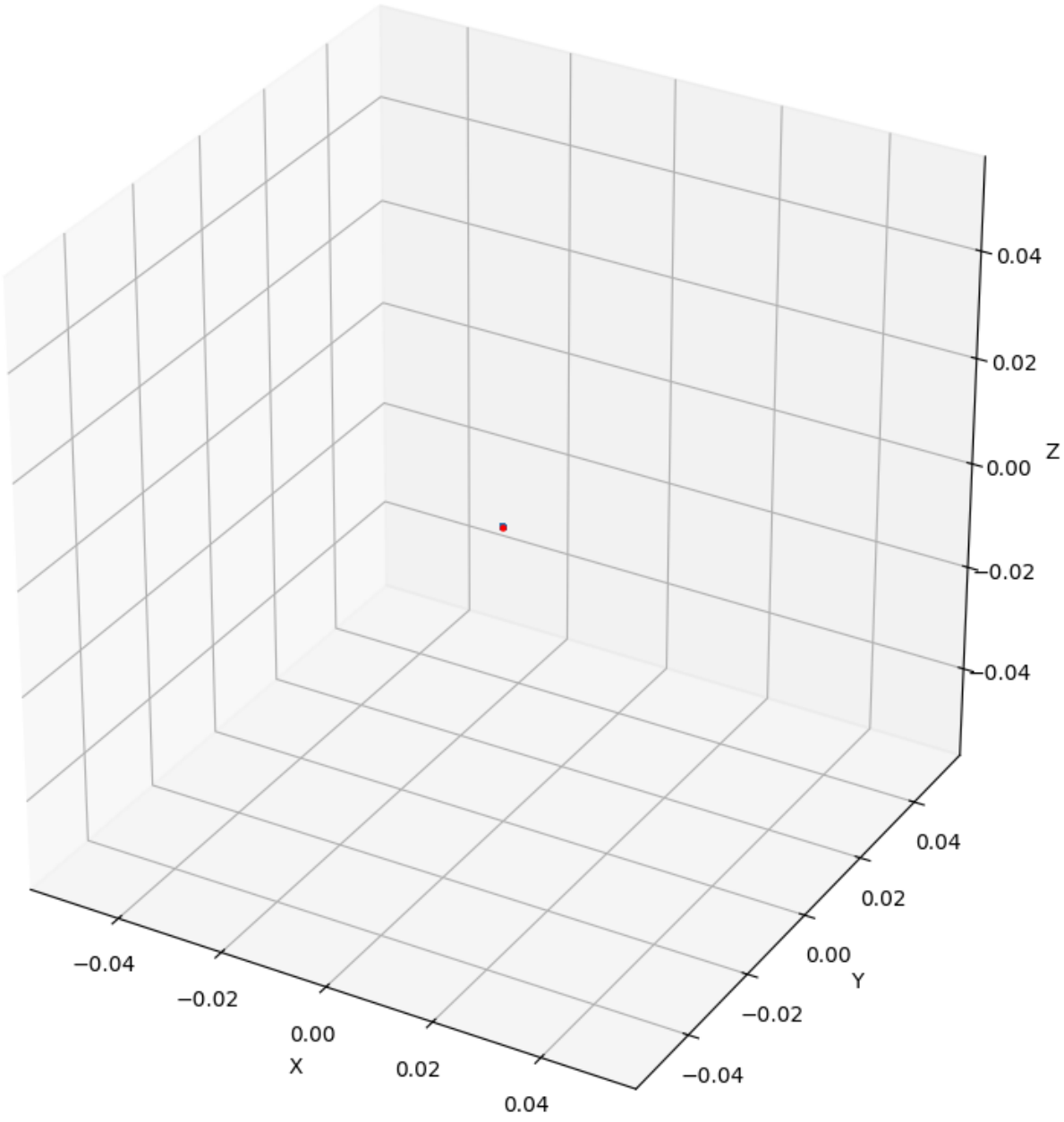
```
With orientation estimation:
length: 0.0
width: 0.0
height: 0.0
orientation: [1. 0. 0.]

Without orientation estimation:
length: 0.0
width: 0.0
height: 0.0
orientation: [1. 0. 0.]
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn.cluster import DBSCAN
from sklearn.preprocessing import MinMaxScaler

def PlotSetting(ax):
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    ax.set_box_aspect([1,1,1])

def GetColors(labels):
    colors = ['tab:blue', 'tab:orange', 'tab:green', 'tab:red', 'tab:purple'
    if isinstance(labels, np.ndarray):
        return np.array(colors)[labels % len(colors)]
    return colors[labels % len(colors)]

def PlotData(ax, points, color='tab:blue'):
    ax.scatter(points[:,0], points[:,1], points[:,2], c=color, marker='.', a

def PlotEllipsoid(ax, center, orientation, color='dodgerblue', npoints=100):
    u = np.linspace(0.0, 2.0 * np.pi, npoints)
    v = np.linspace(0.0, np.pi, npoints)
    x = np.outer(np.cos(u), np.sin(v))
    y = np.outer(np.sin(u), np.sin(v))
    z = np.outer(np.ones_like(u), np.cos(v))

```

```
    for i in range(len(x)):
        for j in range(len(x)):
            x[i,j], y[i,j], z[i,j] = center + np.dot(orientation, [x[i,j], y

    ax.plot_wireframe(x, y, z, rstride=10, cstride=10, color=color, alpha=0.

def PlotArrow(ax, center, orientation, extension=np.ones(3)/2):
    ax.quiver(center[0], center[1], center[2],
              orientation[0][0], orientation[0][1], orientation[0][2],
              length=extension[0], color='tab:blue', normalize=True)
    ax.quiver(center[0], center[1], center[2],
              orientation[1][0], orientation[1][1], orientation[1][2],
              length=extension[1], color='tab:orange', normalize=True)
    ax.quiver(center[0], center[1], center[2],
              orientation[2][0], orientation[2][1], orientation[2][2],
              length=extension[2], color='tab:green', normalize=True)

def PlotClusterEllipsoid(ax, clusterlist, orientation=True, plot_arrow=True,
    cluster_center = ClusterCenter(clusterlist)
    cluster_extension, cluster_orientation = ClusterExtension(clusterlist, c
    print('Center=', cluster_center, '\tL=', cluster_extension[0],
          '\tW=', cluster_extension[1], '\tH=', cluster_extension[2],
          '\tDirection=', cluster_orientation[np.argmax(cluster_extension)])

    ax.scatter(cluster_center[0], cluster_center[1], cluster_center[2], colo
    PlotEllipsoid(ax, cluster_center, (cluster_orientation * HorToVer(cluste

    if plot_arrow:
        PlotArrow(ax, cluster_center, cluster_orientation, cluster_extensior

# Perform clustering
scaler = MinMaxScaler()
scaled_points = scaler.fit_transform(window_points)
dbscan = DBSCAN(eps=0.1, min_samples=5)
clustering = dbscan.fit(scaled_points)

# Create visualization
fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(111, projection='3d')
PlotSetting(ax)

cluster_mask = clustering.labels_ == 0
clusterlist = pd.DataFrame(window_points[cluster_mask], columns=['x', 'y', '
points = np.array(clusterlist)
labels = clustering.labels_[cluster_mask]

PlotData(ax, points, color=GetColors(labels))

unique_labels = np.unique(labels)
for label in unique_labels:
    if label != -1:
```

```
        mask = labels == label
        cluster_points = pd.DataFrame(points[mask], columns=['x', 'y', 'z'])
        PlotClusterEllipsoid(ax, cluster_points, orientation=False, plot_arr

plt.show()
```

```
Center= [0. 0. 0.]      L= 0.0  W= 0.0  H= 0.0  Direction= [1. 0. 0.]
```



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

# Convert frames data to a single DataFrame for visualization
frames_data = []
for i in range(len(data['train'])):
    frame_data = pd.DataFrame(data['train'][i]['new_x'], columns=['x', 'y',
    frames_data.append(frame_data)
datalist = pd.concat(frames_data, ignore_index=True)

# Perform clustering
dbscan = DBSCAN(eps=0.4, min_samples=10)
clustering = dbscan.fit(np.array(datalist[['x', 'y', 'z']]))

# Create visualization
fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(111, projection='3d')
PlotSetting(ax)

# Plot all points with cluster colors
points = np.array(datalist[['x', 'y', 'z']])
PlotData(ax, points, color=GetColors(clustering.labels_))

# Plot ellipsoids for each cluster
for i in range(np.max(np.unique(clustering.labels_))+1):
    cluster_points = datalist[clustering.labels_==i]
    PlotClusterEllipsoid(ax, cluster_points, orientation=True, plot_arrow=Tr

# Plot noise points
PlotData(ax, points[clustering.labels_==-1], color='k')

plt.show()
```

Cluster-based Observation state

Observation state of frame_num = k, cluster_num = n:

$$z(k,n) = [cluster\_center, cluster\_extension, cluster\_orientation]$$

All info of frame_num = k, cluster_num = n:

$$Z(k,n) = [z(k,n), z\_cov(k,n), \{points\_index\}]$$

```
import pandas as pd
import numpy as np
from sklearn.cluster import DBSCAN

def GenerateObservationStates(data_dict, startFrame, endFrame, orientation=T
```

```
    observation_states = InitObservationDataFrame()

    # Convert train data to DataFrame
    frames_data = []
    for i in range(len(data_dict['train'])):
        frame_data = pd.DataFrame(data_dict['train'][i]['new_x'], columns=['
        frame_data['frame_num'] = i
        frames_data.append(frame_data)
    data = pd.concat(frames_data, ignore_index=True)

    for frame_num in range(startFrame, endFrame+1):
        dbscan = DBSCAN(eps=0.4, min_samples=10)
        datalist = data[(data.frame_num >= frame_num) & (data.frame_num < fr

        clustering = ModelCluster(datalist, dbscan, sample_weight=True, show

        for cluster_num in range(-1, np.max(np.unique(clustering.labels_))+1
            clusterlist = datalist[clustering.labels_ == cluster_num]

            observation_state = InitObservationDataFrame()
            observation_state.at[0, 'frame_num'] = frame_num
            observation_state.at[0, 'cluster_num'] = cluster_num
            observation_state.at[0, 'track_num'] = cluster_num
            observation_state.at[0, 'pts'] = list(clusterlist.index)

            if cluster_num != -1:
                cluster_center = ClusterCenter(clusterlist)
                cluster_extension, cluster_orientation = ClusterExtension(cl
                cluster_pt_weight = Weight(clusterlist)
                cluster_cov = CalcCovariance(clusterlist, cluster_center, cl

                observation_state.at[0, ['x','y','z']] = cluster_center
                observation_state.at[0, ['l','w','h']] = cluster_extension
                observation_state.at[0, ['ori_x','ori_y','ori_z']] = cluster
                observation_state.at[0, 'cov'] = [cluster_cov]

            observation_states = pd.concat([observation_states, observation_

        print('Frame', frame_num, 'is done.')

    return observation_states

# Example usage:
observation_states = GenerateObservationStates(data, 0, 10, orientation=True
print(observation_states)
```

```
Frame 0 is done.
Frame 1 is done.
Frame 2 is done.
Frame 3 is done.
Frame 4 is done.
Frame 5 is done.
Frame 6 is done.
Frame 7 is done.
Frame 8 is done.
Frame 9 is done.
Frame 10 is done.
```

	frame_num	cluster_num	track_num	x	y	z	l	w	h	ori_x	ori_
0	0	−1	−1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
1	1	−1	−1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
2	2	−1	−1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
3	3	−1	−1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
4	4	−1	−1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
5	5	−1	−1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
6	6	−1	−1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
7	7	−1	−1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
8	8	−1	−1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
9	9	−1	−1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
10	10	−1	−1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na

	ori_z	cov	pts
0	NaN	NaN	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,...
1	NaN	NaN	[110, 111, 112, 113, 114, 115, 116, 117, 118, ...
2	NaN	NaN	[220, 221, 222, 223, 224, 225, 226, 227, 228, ...
3	NaN	NaN	[330, 331, 332, 333, 334, 335, 336, 337, 338, ...
4	NaN	NaN	[440, 441, 442, 443, 444, 445, 446, 447, 448, ...
5	NaN	NaN	[550, 551, 552, 553, 554, 555, 556, 557, 558, ...
6	NaN	NaN	[660, 661, 662, 663, 664, 665, 666, 667, 668, ...
7	NaN	NaN	[770, 771, 772, 773, 774, 775, 776, 777, 778, ...
8	NaN	NaN	[880, 881, 882, 883, 884, 885, 886, 887, 888, ...
9	NaN	NaN	[990, 991, 992, 993, 994, 995, 996, 997, 998, ...
10	NaN	NaN	[1100, 1101, 1102, 1103, 1104, 1105, 1106, 110...

Track-based estimated state

Estimated state of frame_num = k, track_num = t:
$s(k,t) = [position, velocity, extension, orientation]$

All info of frame_num = k, track_num = t: $S(k,t) = [s(k,t), s\_cov(k,t), Z(k,t)]$

Error covariance of $s(k,t)$: $s\_cov(k,t) = P(k,t)$ in KF updates (later)

```
def InitEstimateDataFrame():
    return pd.DataFrame(columns=['frame_num','track_num','x','y','z','vx','v
```