Tracking in video

Name: Radu-Andrei Bourceanu

This notebook will be used for object tracking for a dataset of LiDAR points on a highway.

For the purpose of this project we will use the dataset provided in the folder `pointclouds`. The dataset consists of 369 .csv files, each one representig a single time frame of a video. Each .csv file consists of 3 columns, x, y, z, that represent the spatial coordinates for each point in the LiDAR point cloud.

Our final aim is to distinguish distinct object in the LiDAR point cloud and perform an object tracking from one frame to another of the object, so that we can finally count how many distinct objects (vehicles in our case) appear in the video (set of frames/.csv files).

For this purpose we will implement the following strategy:

1. We first downsample the data so that we can perform faster calculation on it, without losing the object shapes. For that we have used **Vox Grid Sampling**
2. We apply **DBSCAN** to each individual frame for clustering objects in each individual frame
3. We obtain for each object the centroids and their boundaries so that we can draw the bounding boxes
4. Finally we correlate the clusters from one frame to another and assign each individual object an unique ID so that we can count the total number of trucks
5. We plot the obtained data

RAM cleanup

In [192…

```
# Delete selected variables
'''
del df  # select here variables to delete
import gc
gc.collect()
'''

# OR
```

```
# Reset kernel => delete all variables
%reset
```

Imports

In [193…

```
import pandas as pd
import numpy as np

import os
import glob
import time
from typing import Optional
import warnings

import plotly.express as px
import plotly.graph_objs as go

import hdbscan
from sklearn.cluster import DBSCAN

import math

# Temporarily ignore the 'overflow encountered in cast' RuntimeWarning
# We will encounter a lot this error, as we are reducing the data to float16 from float64, an important aspect to keep in mind
warnings.filterwarnings("ignore", message="overflow encountered in cast")
```

Dataset

In [194…

```
# --- 1. Read all dataframes from the folder efficiently ---
folder_path = 'pointclouds'

# Use glob to find all CSV files in the folder and sort them to maintain order
all_files = sorted(glob.glob(os.path.join(folder_path, "*.csv")))

# Use a list comprehension for an efficient way to read and prepare files
list_of_dfs = []
for i, file_path in enumerate(all_files):
```

```
    df_temp = pd.read_csv(
        file_path,
        usecols=['x', 'y', 'z'],
        dtype={'x': 'float16', 'y': 'float16', 'z': 'float16'}
    )
    # Add the 'time_frame' column to identify each frame
    df_temp['time_frame'] = i
    list_of_dfs.append(df_temp)

# Combine all dataframes into a single one. This is done only once.
print(f"Combining {len(all_files)} files into a single DataFrame...")
df_combined = pd.concat(list_of_dfs, ignore_index=True)
print("DataFrame combined successfully.")
```

```
Combining 369 files into a single DataFrame...
DataFrame combined successfully.
```

Frames filtering

Select only some frames with `time_step` for faster computation during testing

In [195…

```
time_step = 15
df_short = df_combined[df_combined['time_frame'].isin(list(range(0,len(df_combined['time_frame']),time_step)))]
#df_short = df_combined[df_combined['time_frame'].isin(list(range(160,215,1)))]
print(f"The original dataset contained {df_combined['time_frame'].nunique()} frames and the new one contains {df_short['time_f
print(f"The dataste was reduced by {np.round(1-df_short['time_frame'].nunique()/df_combined['time_frame'].nunique(), decimals=
df_short
```

```
The original dataset contained 369 frames and the new one contains 25 frames.
The dataste was reduced by 93.22 %.
```

Out[195…

	x	y	z	time_frame
0	-30.937500	1.330078	2.847656	0
1	-30.968750	1.344727	2.865234	0
2	-31.015625	1.333008	2.855469	0
3	-31.046875	1.333984	2.859375	0
4	-31.093750	1.333008	2.861328	0
...	...	...	...	...
5941783	-24.234375	3.214844	0.302979	360
5941784	-25.671875	3.212891	0.407471	360
5941785	-25.671875	3.171875	0.305420	360
5941786	-25.687500	3.138672	0.225708	360
5941787	-25.671875	3.216797	0.276123	360

411205 rows × 4 columns

Downsampling

- We use Voxel Grid Downsampling for reducing datasize and obtain faster computation times
- A voxel_size of 0.1 reduces data by 80%

In [196…

```
def _voxel_downsample_single_frame(frame_df, voxel_size):
    """
    Helper function to downsample a single frame's point cloud.
    """
    # Create voxel indices for each point in the frame
    voxel_indices = (frame_df[['x', 'y', 'z']] / voxel_size).astype(int)

    # Keep only the first point encountered for each unique voxel index
```

```
    return frame_df.loc[voxel_indices.drop_duplicates().index]

def downsample_point_cloud(full_df, voxel_size=0.1):
    """
    Applies Voxel Grid Downsampling to each time frame in a point cloud DataFrame.

    Args:
        full_df (pd.DataFrame): The complete DataFrame containing all time frames.
                                Must have 'x', 'y', 'z', and 'time_frame' columns.
        voxel_size (float): The side length of the voxel cube.

    Returns:
        pd.DataFrame: The final downsampled DataFrame containing all time frames.
    """
    print(f"Applying Voxel Grid Downsampling with voxel size {voxel_size}...")

    # Group the DataFrame by 'time_frame' and apply the helper function
    # to each individual frame's sub-DataFrame. This is highly efficient.
    downsampled_df = full_df.groupby('time_frame', group_keys=False).apply(
        lambda df: _voxel_downsample_single_frame(df, voxel_size)
    )
    return downsampled_df.reset_index(drop=True)


print(f"Input DataFrame size: {len(df_short):,} points")
with warnings.catch_warnings():
    # We ignore a future warning as it does not apply to our goal
    warnings.simplefilter("ignore", FutureWarning)
    # Perform downsampling
    df_downsampled = downsample_point_cloud(df_short, voxel_size=0.2)
print(f"Downsampled DataFrame size: {len(df_downsampled):,} points")
print(f'Reduction of dataset size by {np.round(1-len(df_downsampled)/len(df_short), decimals=4)*100}%')
print(f"Reduction from orginal dataset by {np.round(1-(df_short['time_frame'].nunique()/df_combined['time_frame'].nunique())*
                                                    (len(df_downsampled)/len(df_short)), decimals=3)*100}%")
df_downsampled
```

```
Input DataFrame size: 411,205 points
Applying Voxel Grid Downsampling with voxel size 0.2...
Downsampled DataFrame size: 84,011 points
Reduction of dataset size by 79.57%
Reduction from orginal dataset by 98.6%
```

Out[196…

	x	y	z	time_frame
0	-30.937500	1.330078	2.847656	0
1	-31.015625	1.333008	2.855469	0
2	-31.203125	1.332031	2.869141	0
3	-31.390625	1.330078	2.884766	0
4	-31.609375	1.236328	2.814453	0
...	...	...	...	...
84006	-25.781250	3.226562	0.601074	360
84007	-25.828125	3.210938	0.567871	360
84008	-23.890625	3.203125	0.246704	360
84009	-24.234375	3.214844	0.302979	360
84010	-25.671875	3.216797	0.276123	360

84011 rows × 4 columns

Plot

In the following code, we create a function for plotting the frames using plotly.graph_objects

In [197…

```
df = df_downsampled

def show_video(df: Optional[pd.DataFrame] = None, obj_df: Optional[pd.DataFrame] = None, box_list: Optional[list] = None, zoom

    start_time = time.perf_counter()

    print("Generating animated plot with Plotly Graph Objects...")

    #### 1. Create the Figure ----------------------------------------------------------------------------------------
```

```
    fig = go.Figure()

    ### 2. Check if clusters exist and generate time frames-----------------------------------------------------------------

    time_frames = []

    # For LiDAR points
    if df is not None:
        has_clusters = 'cluster' in df.columns
        color_map = {}
        if has_clusters:
            # Create a custom color map only if the 'cluster' column exists
            unique_labels = sorted([label for label in df['cluster'].unique() if pd.notna(label) and label != -1])
            color_sequence = px.colors.qualitative.Dark24
            color_map = {**{str(label): color_sequence[i] for i, label in enumerate(unique_labels)}}

        time_frames = sorted(df['time_frame'].unique())

    # For bounding boxes
    if (not time_frames) and (obj_df is not None):
        time_frames = sorted(obj_df['time_frame'].unique())

    ### 3. Add a trace for each time frame-----------------------------------------------------------------------------------

    frame_plot = 0
    for frame_id in time_frames:

        # For LiDAR points
        if df is not None:
            frame_df = df[df['time_frame'] == frame_id]

            if has_clusters:
                marker_colors = [color_map.get(str(c), 'lightgrey') for c in frame_df['cluster']]
            else:
                marker_colors = 'cornflowerblue'  # Use a single default color

            scatter_trace = go.Scatter3d(x=frame_df['x'], y=frame_df['y'], z=frame_df['z'], mode='markers', marker=dict(size=1
                                         visible=(frame_id == time_frames[0]), # Only the first frame is visible initially
                                         name=f'LiDAR points')
        # For bounding boxes
        if obj_df is not None:
```

```
            box_trace = go.Scatter3d(x=box_list[frame_plot][0], y=box_list[frame_plot][1], z=box_list[frame_plot][2], mode='li
            visible=(frame_id == time_frames[0]), # Only the first frame is visible initially
            name=f'Bounding boxes')

        # For LiDAR points
        if df is not None:
            fig.add_trace(scatter_trace)
        # For bounding boxes
        if obj_df is not None:
            fig.add_trace(box_trace)

        frame_plot = frame_plot +1

    ### 4. Create the slider and animation control--------------------------------------------------------------------------

    steps = []
    num_traces_per_frame = (1 if df is not None else 0) + (1 if obj_df is not None and box_list is not None else 0)

    for i, frame_id in enumerate(time_frames):
        step = dict(
            method="update",
            args=[{"visible": [False] * len(fig.data)}],
            label=str(frame_id)
        )
        # Make all traces for the current frame visible
        start_index = i * num_traces_per_frame
        for j in range(num_traces_per_frame):
            if start_index + j < len(fig.data):
                step["args"][0]["visible"][start_index + j] = True
        steps.append(step)

    sliders = [dict(active=0, currentvalue={"prefix": "Time Frame: "}, pad={"t": 50}, steps=steps)]

    ### 5. Customize the final layout ------------------------------------------------------------------------------------

    zoom_factor = 1 / zoom

    detected_veh = 0
    # Detected vehicles
    if (df is not None) and has_clusters:
        detected_veh = (df['cluster'].max()+1).astype(int)
```

```
    if obj_df is not None:
        detected_veh = (obj_df['new_object_id'].max()+1).astype(int)

    fig.update_layout(sliders=sliders,
                      title=dict(text=f"3D Object Detection. Detected vehicles in video: {detected_veh}", x=0.5, y=0.95,
                                 font= dict(family='Arial, sans-serif', size=28, color='black'), xanchor='center', yanchor=
                      scene=dict( xaxis=dict(title='X', range=[-70 * zoom_factor, 30 * zoom_factor]),
                                  yaxis=dict(title='Y', range=[-10 * zoom_factor, 10 * zoom_factor]),
                                  zaxis=dict(title='Z', range=[-10 * zoom_factor, 10 * zoom_factor]),
                                  aspectmode='manual',
                                  aspectratio=dict(x=10, y=2, z=2),
                                  camera=dict(
                          # Position the camera along the Y-axis
                          eye=dict(x=-0.5, y=5, z=2)
                      )
                      ),
        width=1400, height=600,)

    ### 6. Show the figure and computation time ------------------------------------------------------------------------

    fig.show()

    end_time = time.perf_counter()
    duration = end_time - start_time
    print(f'Computing the plot for {len(time_frames)} time frames took {np.round(duration, decimals=2)} seconds.\n')

    ### END of function-------------------------------------------------------------------------------------------------

show_video(df, zoom = 1, opacity=1)
```

```
Generating animated plot with Plotly Graph Objects...
Computing the plot for 25 time frames took 0.59 seconds.
```

Clustering: DBSCAN

CLuster the objects in the frames using DBSCAN, for performing object detection

In [198…
```
df = df_downsampled

def cluster_dbscan(frame_df, min_samples=15, eps=0.5):
    frame_df['cluster'] = np.nan

    frames = frame_df['time_frame'].unique()
    for t in frames:
        points = frame_df.loc[frame_df['time_frame'] == t, ['x', 'y', 'z']]
        clusterer = DBSCAN(eps=eps, min_samples=min_samples)
        labels = clusterer.fit_predict(points)
        frame_df.loc[df_downsampled['time_frame'] == t, 'cluster'] = labels

# Clustering (initial good values: min_samples=40, eps=3.5)
print("Starting clustering")
start_time = time.perf_counter()
cluster_dbscan(df, min_samples=40, eps=3.5)
end_time = time.perf_counter()
duration = end_time - start_time
print(f'Computing HDBSCAN for {df['time_frame'].nunique()} time frames took {np.round(duration, decimals=2)} seconds.\n')
#print(f'The dataframe is {df}')

df_dbscan = df

show_video(df, zoom = 1, opacity=1)
```

```
Starting clustering
Computing HDBSCAN for 25 time frames took 2.61 seconds.

Generating animated plot with Plotly Graph Objects...
Computing the plot for 25 time frames took 3.56 seconds.
```

Centroids and min/max corners

Calculate for each object in each frame its centroid and min/max corner

In [199…
```
#Define the df used for this code
df = df_dbscan
```

```python
#-------------------------------------------------------------------------------------------
def calculate_bounding_boxes(clustered_df):
    """
    Calculates the Axis-Aligned Bounding Box (AABB) and centroid for each
    object cluster in a DataFrame.

    Args:
        clustered_df (pd.DataFrame): DataFrame containing point cloud data.
                                     Must have 'x', 'y', 'z', and 'cluster' columns.

    Returns:
        list: A list of dictionaries. Each dictionary represents one detected
              object and contains its 'cluster_id', 'centroid', 'min_corner',
              and 'max_corner'.
    """

    detected_objects = []

    # Get all unique cluster IDs, excluding -1 (noise)
    unique_cluster_ids = clustered_df['cluster'].unique()
    # Uncomment next row if you want to filter out the noise cluster
    unique_cluster_ids = [cid for cid in unique_cluster_ids if cid != -1 and not pd.isna(cid)]

    # Iterate through each unique cluster
    for cluster_id in unique_cluster_ids:
        # Get all points belonging to the current cluster
        cluster_points = clustered_df[clustered_df['cluster'] == cluster_id]

        if not cluster_points.empty:
            # Calculate the min and max corners of the bounding box
            min_corner = cluster_points[['x', 'y', 'z']].min().values
            max_corner = cluster_points[['x', 'y', 'z']].max().values

            # Calculate the centroid (average position) of the cluster
            # A temporary copy is first produced as float 64 to avoid overflow problems in the sum of mean () method and after
            centroid = cluster_points[['x', 'y', 'z']].astype('float64').mean().values.astype('float16')

            # Store the object's information
            detected_objects.append({
                'cluster_id': cluster_id,
                'centroid': centroid,
```

```
                'min_corner': min_corner,
                'max_corner': max_corner
            })

    return detected_objects

obj_list = []

for frame in df['time_frame'].unique():
    #print(f"Calculating parameters for frame {frame}")
    objects = calculate_bounding_boxes(df[df['time_frame'] == frame])
    for obj in objects:
        #print(f"""--- Object Cluster ID: {obj['cluster_id']}. Centroid: {np.round(obj['centroid'], 2)}. Min Corner (x,y,z): {

        # Collect all the data in a list
        new_row = {
            'centroid': np.round(obj['centroid'], 2),
            'min_corner': np.round(obj['min_corner'], 2),
            'max_corner': np.round(obj['max_corner'], 2),
            'object_id': obj['cluster_id'],
            'time_frame': frame
        }
        obj_list.append(new_row)
    #print("-" * 60)


# Convert list to a df
columns = ['centroid', 'min_corner', 'max_corner', 'object_id', 'time_frame']
obj_df = pd.DataFrame(obj_list, columns=columns)
obj_df_init = obj_df
obj_df_init.head(10)
```

Out[199…

	centroid	min_corner	max_corner	object_id	time_frame
0	[-33.25, 2.56, 1.95]	[-40.8, 0.92, 0.2]	[-29.12, 3.61, 3.11]	0.0	0
1	[-15.95, 1.79, 1.26]	[-23.4, 0.21, 0.2]	[-4.8, 3.68, 3.98]	1.0	0
2	[-53.1, 2.11, 2.29]	[-55.0, 0.51, 0.22]	[-52.38, 3.5, 4.03]	2.0	0
3	[-31.42, 2.56, 1.91]	[-37.84, 1.0, 0.2]	[-27.08, 3.62, 3.1]	0.0	15
4	[-14.2, 1.73, 1.26]	[-21.55, 0.2, 0.2]	[-2.04, 3.69, 3.94]	1.0	15
5	[-51.44, 2.23, 2.48]	[-54.97, 0.53, 0.2]	[-50.12, 3.5, 4.16]	2.0	15
6	[-30.19, 2.58, 1.93]	[-35.4, 1.02, 0.2]	[-25.58, 3.5, 3.12]	0.0	30
7	[-13.19, 1.68, 1.26]	[-20.31, 0.21, 0.24]	[-2.63, 3.69, 3.95]	1.0	30
8	[-50.28, 2.3, 2.53]	[-54.9, 0.51, 0.21]	[-48.4, 3.45, 4.16]	2.0	30
9	[-28.86, 2.58, 1.98]	[-33.22, 0.94, 0.2]	[-24.12, 3.5, 3.12]	0.0	45

Bounding Box Calculation

Based on the min/max corners, calculate the bounding boxes for each object

In [200…

```python
# In this cell we make use only of obj_df calculated in the anterior cell

# Function for obtaining the bounding boxes ----------------------------------------------------------------------------
def get_bounding_box_lines(min_corner, max_corner):
    """
    Generates the X, Y, Z coordinates for the 12 lines of an AABB.
    This revised version is more robust and ensures correct line drawing.
    """
    x_min, y_min, z_min = min_corner
    x_max, y_max, z_max = max_corner

    # Define the 8 corners of the bounding box
    corners = [
        [x_min, y_min, z_min],  # Corner 0
```

```
        [x_max, y_min, z_min],  # Corner 1
        [x_max, y_max, z_min],  # Corner 2
        [x_min, y_max, z_min],  # Corner 3
        [x_min, y_min, z_max],  # Corner 4
        [x_max, y_min, z_max],  # Corner 5
        [x_max, y_max, z_max],  # Corner 6
        [x_min, y_max, z_max]   # Corner 7
    ]

    # Define the 12 lines by connecting pairs of corner indices
    line_indices = [
        (0, 1), (1, 2), (2, 3), (3, 0),  # Bottom face
        (4, 5), (5, 6), (6, 7), (7, 4),  # Top face
        (0, 4), (1, 5), (2, 6), (3, 7)   # Vertical edges connecting the faces
    ]

    # Create the coordinate lists for plotting
    x_lines, y_lines, z_lines = [], [], []

    for start_idx, end_idx in line_indices:
        # Get the start and end points of the line
        p1 = corners[start_idx]
        p2 = corners[end_idx]

        # Add the coordinates to the lists
        x_lines.extend([p1[0], p2[0], None]) # Append start, end, and None to break the line
        y_lines.extend([p1[1], p2[1], None])
        z_lines.extend([p1[2], p2[2], None])

    #return x_lines, y_lines, z_lines
    return x_lines, y_lines, z_lines
#-------------------------------------------------------------------------------------

# Rows = time frame
# Columns = x, y, z input for lines in plot
box_list = [[np.nan for _ in range(3)] for _ in range(obj_df['time_frame'].nunique())]
frame_no = 0
for frame in obj_df['time_frame'].unique():
    x_lines = []
    y_lines = []
    z_lines = []
```

```
    for cluster_id in obj_df.loc[obj_df['time_frame'] == frame, 'object_id']:

        min_corner = obj_df.loc[(obj_df['time_frame'] == frame) & (obj_df['object_id'] == cluster_id), 'min_corner'].iloc[0]
        max_corner = obj_df.loc[(obj_df['time_frame'] == frame) & (obj_df['object_id'] == cluster_id), 'max_corner'].iloc[0]
        box_lines = get_bounding_box_lines(min_corner, max_corner)
        x_lines.extend(box_lines[0])
        y_lines.extend(box_lines[1])
        z_lines.extend(box_lines[2])
    box_list[frame_no][0] = x_lines
    box_list[frame_no][1] = y_lines
    box_list[frame_no][2] = z_lines
    frame_no = frame_no + 1

# Create a new column for object_id's that will be changed in the future (column necessary for plotting function)
obj_df['new_object_id'] = obj_df['object_id']
obj_df_init = obj_df

show_video(obj_df=obj_df, box_list=box_list, zoom = 1, opacity=1)
```

```
Generating animated plot with Plotly Graph Objects...
Computing the plot for 25 time frames took 0.17 seconds.
```

Plot bounded trucks

In [201…

```
df = df_dbscan
show_video(df, obj_df=obj_df, box_list=box_list, zoom = 1, opacity=1)
```

```
Generating animated plot with Plotly Graph Objects...
Computing the plot for 25 time frames took 4.18 seconds.
```

Cluster ID correlation

- Finally we need to correlate the object from one frame to another
- We do this by correlating their centroids

In [202…

```
df = df_dbscan
obj_df = obj_df_init

obj_df['new_object_id'] = np.nan
obj_df.loc[obj_df['time_frame'] == 0, 'new_object_id'] = obj_df.loc[obj_df['time_frame'] == 0, 'object_id']

threshold = 15
cluster_max = 0
# List with all the frame values
time_frames = obj_df['time_frame'].unique()

for frame in list(range(1,len(time_frames))):
    # Calculate max cluster ID from previous frame, to check if new cluster appears in current frame
    cluster_max = obj_df.loc[obj_df['time_frame'] == time_frames[frame-1], 'new_object_id'].max()
    # for-loop for iterating through clusters of the current frame
    for j in obj_df.loc[obj_df['time_frame'] == time_frames[frame], 'object_id']:
        index = np.inf
        # for-loop for iterating through clusters of the previous time frame
        for i in obj_df.loc[obj_df['time_frame'] == time_frames[frame-1], 'new_object_id']:

            # Coordinates of cluster from previous time step-
            x_previous = obj_df.loc[(obj_df['time_frame'] == time_frames[frame-1]) & (obj_df['new_object_id'] == i), 'centroid
            y_previous = obj_df.loc[(obj_df['time_frame'] == time_frames[frame-1]) & (obj_df['new_object_id'] == i), 'centroid
            z_previous = obj_df.loc[(obj_df['time_frame'] == time_frames[frame-1]) & (obj_df['new_object_id'] == i), 'centroid

            # Coordinates of cluster from current time step
            x_current = obj_df.loc[(obj_df['time_frame'] == time_frames[frame]) & (obj_df['object_id'] == j), 'centroid'].iloc
            y_current = obj_df.loc[(obj_df['time_frame'] == time_frames[frame]) & (obj_df['object_id'] == j), 'centroid'].iloc
            z_current = obj_df.loc[(obj_df['time_frame'] == time_frames[frame]) & (obj_df['object_id'] == j), 'centroid'].iloc

            # Calculate distance between the clusters
            distance = math.sqrt((x_current - x_previous)**2 + (y_current - y_previous)**2 + (z_current - z_previous)**2)

            # Matching centroids, we obtain the corresponding centroid from previous step
            if (distance < threshold):
                index = i #make something here for the assignment for cluster 3

        # Case when something needs to be switched
        if index != np.inf:
            obj_df.loc[(obj_df['time_frame'] == time_frames[frame]) & (obj_df['object_id'] == j), 'new_object_id'] = index
```

```
        else:
            obj_df.loc[(obj_df['time_frame'] == time_frames[frame]) & (obj_df['object_id'] == j), 'new_object_id'] = cluster_m

obj_df
```

Out[202…

	centroid	min_corner	max_corner	object_id	time_frame	new_object_id
0	[-33.25, 2.56, 1.95]	[-40.8, 0.92, 0.2]	[-29.12, 3.61, 3.11]	0.0	0	0.0
1	[-15.95, 1.79, 1.26]	[-23.4, 0.21, 0.2]	[-4.8, 3.68, 3.98]	1.0	0	1.0
2	[-53.1, 2.11, 2.29]	[-55.0, 0.51, 0.22]	[-52.38, 3.5, 4.03]	2.0	0	2.0
3	[-31.42, 2.56, 1.91]	[-37.84, 1.0, 0.2]	[-27.08, 3.62, 3.1]	0.0	15	0.0
4	[-14.2, 1.73, 1.26]	[-21.55, 0.2, 0.2]	[-2.04, 3.69, 3.94]	1.0	15	1.0
…	…	…	…	…	…	…
70	[-42.47, 2.23, 2.78]	[-52.72, 0.26, 0.21]	[-36.9, 3.14, 4.12]	1.0	345	3.0
71	[3.51, 1.18, 1.4]	[3.28, 0.22, 0.23]	[6.8, 2.2, 2.81]	2.0	345	0.0
72	[-20.72, 2.57, 3.01]	[-26.55, 0.52, 0.2]	[-9.89, 3.34, 4.18]	0.0	360	2.0
73	[-41.2, 2.26, 2.82]	[-51.03, 0.28, 0.2]	[-35.06, 3.1, 4.12]	1.0	360	3.0
74	[8.76, 1.19, 0.99]	[8.53, 0.21, 0.24]	[10.84, 3.67, 2.07]	2.0	360	0.0

75 rows × 6 columns

In [203…

```
mapping = obj_df[['time_frame', 'object_id', 'new_object_id']].drop_duplicates()

tracking_df = df.merge(
    mapping,
    how='left',
    left_on=['time_frame', 'cluster'],
    right_on=['time_frame', 'object_id']
)

# Replace cluster with mapped new_object_id where available
tracking_df['cluster'] = tracking_df['new_object_id'].combine_first(tracking_df['cluster'])
```

```
# Drop unnecesary columns
tracking_df = tracking_df.drop(columns=['object_id', 'new_object_id'])

show_video(tracking_df, obj_df=obj_df, box_list=box_list, zoom = 1, opacity=1)
```

```
Generating animated plot with Plotly Graph Objects...
Computing the plot for 25 time frames took 4.9 seconds.
```