In [1]:
```python
import os
import time
import json
import numpy as np
import pandas as pd
import scipy.ndimage as ndimage
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
from math import atan2, degrees
from PIL import Image, ImageDraw
from scipy.interpolate import interp1d, splprep, splev
from scipy.optimize import minimize
from sklearn.cluster import DBSCAN
from sklearn.linear_model import RANSACRegressor
from matplotlib.animation import FuncAnimation
from matplotlib.patches import Rectangle, FancyArrowPatch
from matplotlib.colors import Normalize
from matplotlib.cm import ScalarMappable
from IPython.display import HTML
import pyproj
from pyproj import Transformer
import time
from scipy.interpolate import splprep, splev
from watchdog.observers import Observer
from watchdog.events import FileSystemEventHandler

# Define the output folder and column names
output_folder = "C:/Users/User/Documents/DATA/Gps"  # Main data folder

column_names = [
    'Record_Type', 'Blank', 'Event_Number', 'Date', 'Time', 'Timestamp', 'Solution_Status',
    'Lat', 'Lng', 'Height', 'Heading[°]', 'Velocity[m/s]', 'Velocity[km/h]', 'Acceleration[m/s²]'
]

class MyHandler(FileSystemEventHandler):
    def on_created(self, event):  # this method is called when a file or directory is created
        print(f'event type: {event.event_type}  path : {event.src_path}')
        import os
        import time
        import json
        import pandas as pd  # <-- Move the pandas import here
        if os.path.isfile(event.src_path) and event.src_path.endswith(".csv"):  # check if the created event is a CSV file
            csv_path = event.src_path  # Full path to the csv file
            csv_name = os.path.splitext(os.path.basename(csv_path))[0]  # Get the name of the csv file without the extension
            parent_folder = os.path.dirname(csv_path)  # Get the parent folder path

            print(f"Processing file: {csv_name} in folder: {parent_folder}")

            # Load CSV data with no header
            df = pd.read_csv(csv_path, delimiter=';', header=None, skiprows=1)

            # Check the number of columns in the DataFrame
            if df.shape[1] <= 3:
                print(f"Skip {csv_path} as it does not have more than 3 columns")
                return  # Skip this file and move to the next one

            # If the CSV file has more than 3 columns, then apply column names
            df.columns = column_names

            # Convert columns to numeric
            df['Lat'] = pd.to_numeric(df['Lat'], errors='coerce')
            df['Lng'] = pd.to_numeric(df['Lng'], errors='coerce')
            df['Height'] = pd.to_numeric(df['Height'], errors='coerce')

            # Drop rows with null values in 'Lat', 'Lng', and 'Height'
            df = df.dropna(subset=['Lat', 'Lng', 'Height'])

            # Define the Projections
            wgs84=pyproj.CRS("EPSG:4326") # WGS84
            UTM=pyproj.CRS("EPSG:32633") # UTM zone 33N for example. Update it as per your requirement.

            # Define transformer
            transformer = Transformer.from_crs(wgs84, UTM, always_xy=True)

            # Create a new DataFrame to store the results
            df_utm = pd.DataFrame()

            # Convert the coordinates
            df_utm['X'], df_utm['Y'], df_utm['Z'] = transformer.transform(df['Lat'].tolist(), df['Lng'].tolist(), df['Height'].tolist())

            # Add the last four columns to df_utm
            df_utm['Heading[°]'] = df['Heading[°]']
            df_utm['Velocity[m/s]'] = df['Velocity[m/s]']
            df_utm['Velocity[km/h]'] = df['Velocity[km/h]']
            df_utm['Acceleration[m/s²]'] = df['Acceleration[m/s²]']

            # Convert the DataFrame to a JSON string
            json_str = df_utm.to_json(orient='records')

            # Create JSON file path in the current subdirectory
            json_name = os.path.splitext(os.path.basename(csv_path))[0]  # Get the name of the csv file without the extension
            json_path = os.path.join(os.path.dirname(csv_path), f'{json_name}.json')

            # Write the JSON string to a file
            with open(json_path, 'w') as f:
                f.write(json_str)

            # Load JSON data
            with open(json_path, "r") as f:
                data = json.load(f)

            # Extract XY coordinates and speed from the trajectory
            trajectory_points = [(record['X'], record['Y']) for record in data]

            # Extract speed values. If 'Velocity[km/h]' does not exist, use None
            speed_values = [record.get('Velocity[km/h]', None) for record in data]

            # Convert trajectory_points to a numpy array
            trajectory_points = np.array([(record['X'], record['Y']) for record in data])

            # Implement DBSCAN on the raw trajectory
            db = DBSCAN(eps=10, min_samples=5).fit(trajectory_points)

            best_score = 0
            best_fit_line = None
            best_segment = None

            # Convert db.labels_ to a numpy array
            labels = np.array(db.labels_)

            # Iterate over the clusters produced by DBSCAN
            for cluster_index in np.unique(labels):
                if cluster_index == -1:
                    continue  # Skip the noise (outlier) cluster
```

```python
        # Extract this cluster
        mask = labels == cluster_index
        x_cluster = trajectory_points[mask, 0]
        y_cluster = trajectory_points[mask, 1]

        # Fit a RANSAC line to this cluster
        ransac_cluster = RANSACRegressor().fit(x_cluster.reshape(-1, 1), y_cluster)

        # Compute the Euclidean distance between the first and last point of the line
        inlier_mask = ransac_cluster.inlier_mask_
        x_cluster_inliers = x_cluster[inlier_mask]
        y_cluster_inliers = y_cluster[inlier_mask]

        # Calculate the Euclidean distance
        distance = np.sqrt((x_cluster_inliers[-1] - x_cluster_inliers[0])**2 + (y_cluster_inliers[-1] - y_cluster_inliers[0])**2)

        # Number of inliers
        num_inliers = inlier_mask.sum()

        # Compute score with 80% weight on distance and 20% weight on number of inliers
        score = 0.8 * distance + 0.2 * num_inliers

        # Update the best line if this line has a higher score
        if score > best_score:
            best_score = score
            best_fit_line = ransac_cluster
            best_segment = np.column_stack((x_cluster_inliers, y_cluster_inliers))

    # Calculate the line's slope and intercept
    slope = best_fit_line.estimator_.coef_[0]
    intercept = best_fit_line.estimator_.intercept_

    print(f"Best Segment Score: {best_score}")
    print(f"Slope: {slope}, Intercept: {intercept}")

    # Assume best_segment contains the best segment identified by DBSCAN and RANSAC
    trajectory_x = best_segment[:, 0]
    trajectory_y = best_segment[:, 1]

    # Apply the Gaussian filter for smoothing/denoising
    smoothed_trajectory_x = ndimage.gaussian_filter(trajectory_x, sigma=5)
    smoothed_trajectory_y = ndimage.gaussian_filter(trajectory_y, sigma=5)

    # Interpolate points to fill gaps with up to 0.1 meters resolution
    interpolator = interp1d(smoothed_trajectory_x, smoothed_trajectory_y, kind='cubic')
    x_interpolated = np.linspace(smoothed_trajectory_x.min(), smoothed_trajectory_x.max(), num=int(np.ptp(smoothed_trajectory_x) / 0.1))
    y_interpolated = interpolator(x_interpolated)

    # Get the first raw point of the longest segment
    first_raw_point = np.array([trajectory_x[0], trajectory_y[0]])
    nearest_index = np.argmin(np.linalg.norm(np.column_stack((smoothed_trajectory_x, smoothed_trajectory_y)) - first_raw_point, axis=1))

    # Update the x_interpolated and y_interpolated arrays to start from the nearest point
    x_interpolated = np.roll(x_interpolated, -nearest_index)
    y_interpolated = np.roll(y_interpolated, -nearest_index)

    # Check if the trajectory is reversed and flip the order of the interpolated points if necessary
    distance_to_first = np.linalg.norm(first_raw_point - np.array((x_interpolated[0], y_interpolated[0])))
    distance_to_last = np.linalg.norm(first_raw_point - np.array((x_interpolated[-1], y_interpolated[-1])))

    if distance_to_last < distance_to_first:
        x_interpolated = np.flip(x_interpolated)
        y_interpolated = np.flip(y_interpolated)

    # Calculate the line's start point (point A)
    point_A = (x_interpolated[0], y_interpolated[0])

    import pandas as pd
    import os

    # Function to calculate signed distance
    def signed_distance_to_line(x_points, y_points, point_A, point_B):
        vec_AB = np.array([point_B[0] - point_A[0], point_B[1] - point_A[1]])
        vec_AP = np.array([x_points - point_A[0], y_points - point_A[1]])

        vec_AB_norm = vec_AB / np.linalg.norm(vec_AB)
        vec_AP_proj = np.dot(vec_AP.T, vec_AB_norm)

        vec_orthogonal = vec_AP - np.outer(vec_AP_proj, vec_AB_norm).T
        signed_distances = np.linalg.norm(vec_orthogonal, axis=0) * np.sign(np.cross(vec_AP.T, vec_AB))

        return signed_distances

    from scipy import stats

    # File path
    file_path = "C:/Users/User/Documents/Data/average_signed_distances.txt"

    # Check if the file exists
    if not os.path.isfile(file_path):
        # If not, create an empty file
        open(file_path, 'w').close()

    # Check if the file is not empty
    if os.stat(file_path).st_size != 0:
        # Read the file into a pandas dataframe
        df = pd.read_csv(file_path, header=None, sep=",")

        # Convert strings to numeric values, non-numeric strings become NaN
        df = df.apply(pd.to_numeric, errors='coerce')

        # Fill NaN values with forward fill, backward fill, then zero as a last resort
        df.fillna(method='ffill', inplace=True)
        df.fillna(method='bfill', inplace=True)
        df.fillna(0, inplace=True)

        # Drop rows with NaN values (there should be none left after the fillna steps)
        df.dropna(inplace=True)

        # Remove duplicates
        df = df.drop_duplicates()

        # Calculate the most frequent positive and negative signed distances, rounded to one decimal place
        most_freq_positive = df[0].round(1).mode()[0]
        most_freq_negative = df[1].round(1).mode()[0]

        # If there are at least two occurrences of the most frequent value, incorporate it into the average
        if (df[0].round(1) == most_freq_positive).sum() >= 2:
            avg_positive = (df[0].mean() + most_freq_positive) / 2
        else:
            avg_positive = df[0].mean()

        if (df[1].round(1) == most_freq_negative).sum() >= 2:
            avg_negative = (df[1].mean() + most_freq_negative) / 2
```

```python
        else:
            avg_negative = df[1].mean()

        # Calculate the max positive and min negative signed distances
        max_positive = df[0].max()
        min_negative = df[1].min()

        # Calculate the minimum and maximum averages
        min_avg = min(avg_negative, min_negative, most_freq_negative)*1.5
        max_avg = max(avg_positive, max_positive, most_freq_positive)*1.5

        print(f"Minimum average: {min_avg}")
        print(f"Maximum average: {max_avg}")
        print(f"Average positive: {avg_positive}")
        print(f"Average negative: {avg_negative}")
        print(f"Maximum positive: {max_positive}")
        print(f"Minimum negative: {min_negative}")
        print(f"Most frequent positive: {most_freq_positive}")
        print(f"Most frequent negative: {most_freq_negative}")
    else:
        # Default values if the file is empty
        min_avg = -1
        max_avg = 1
        print("The file is empty. Default values are used.")
        print(f"Minimum average: {min_avg}")
        print(f"Maximum average: {max_avg}")

    point_A = (x_interpolated[0], y_interpolated[0])
    point_B = (x_interpolated[-1], y_interpolated[-1])

    # Indices of points with signed distances within the range [min_avg, max_avg]
    signed_distances = signed_distance_to_line(x_interpolated, y_interpolated, point_A, point_B)
    valid_indices = np.where((signed_distances >= min_avg) & (signed_distances <= max_avg))


    # Filter the points
    x_filtered = x_interpolated[valid_indices]
    y_filtered = y_interpolated[valid_indices]

    from scipy.ndimage import gaussian_filter

    # Apply Gaussian smoothing
    smoothed_trajectory_x = gaussian_filter(x_filtered, sigma=5)
    smoothed_trajectory_y = gaussian_filter(y_filtered, sigma=5)

    # Interpolate points to fill gaps with up to 0.1 meters resolution
    interpolator = interp1d(smoothed_trajectory_x, smoothed_trajectory_y, kind='cubic')
    x_interpolated = np.linspace(smoothed_trajectory_x.min(), smoothed_trajectory_x.max(), num=int(np.ptp(smoothed_trajectory_x) / 0.1))
    y_interpolated = interpolator(x_interpolated)

    # Generate the best-fit straight line
    def line_error(params):
        a = params[0]
        y_line = a * (x_interpolated - point_A[0]) + point_A[1]
        return np.sum((y_line - y_interpolated)**2)

    best_fit_line = minimize(line_error, [0])
    a = best_fit_line.x[0]
    y_best_fit_line = a * (x_interpolated - point_A[0]) + point_A[1]

    # Calculate the line's start point (point_A_interpolated)
    point_A_interpolated = (x_interpolated[0], y_interpolated[0])

    # Calculate the point B (the end point of the straight line)
    point_B_line = (x_interpolated[-1], y_best_fit_line[-1])

    # Recalculate signed distances from the interpolated points to the straight line
    signed_distances = signed_distance_to_line(x_interpolated, y_interpolated, point_A_interpolated, point_B_line)

    # Calculate average positive and negative signed distances
    avg_positive = np.mean([dist for dist in signed_distances if dist > 0])
    avg_negative = np.mean([dist for dist in signed_distances if dist < 0])

    # Prepare the data to be written
    data_to_write = f"{avg_positive}, {avg_negative}\n"

    # Append the data to the file
    with open(file_path, "a") as file:
        file.write(data_to_write)

    # Calculate the stationing
    stationing = [0]
    for i in range(1, len(x_interpolated)):
        delta_x = x_interpolated[i] - x_interpolated[i - 1]
        delta_y = y_interpolated[i] - y_interpolated[i - 1]
        stationing.append(stationing[-1] + np.sqrt(delta_x**2 + delta_y**2))

    # Calculate the abs of the maximum signed distance and the abs of the minimum signed distance
    min_abs_signed_distance = np.min(np.abs(signed_distances))
    max_abs_signed_distance = np.max(np.abs(signed_distances))

    # Create the heatmap colors
    colors = ['green', 'yellow', 'red']
    norm = mcolors.Normalize(vmin=min_abs_signed_distance, vmax=max_abs_signed_distance)
    cmap = mcolors.LinearSegmentedColormap.from_list('heatmap', colors)
    point_colors = cmap(norm(np.abs(signed_distances)))

    # Export the signed distances and stationing in CSV
    data = {"Stationing": stationing, "Signed Distance": signed_distances}
    data_df = pd.DataFrame(data)
    data_df.to_csv(os.path.join(parent_folder, f"signed_distances_and_stationing-{csv_name}.csv"), index=False)

    # Export the raw trajectory and interpolated trajectory as TXT files
    with open(os.path.join(parent_folder, f"raw_trajectory-{csv_name}.txt"), "w") as f:
        for x, y in zip(trajectory_x, trajectory_y):
            f.write(f"({x}, {y})\n")

    with open(os.path.join(parent_folder, f"interpolated_trajectory-{csv_name}.txt"), "w") as f:
        for x, y in zip(x_interpolated, y_interpolated):
            f.write(f"({x}, {y})\n")

    # Define a similarity threshold
    similarity_threshold = 25  # meters

    # Get the absolute difference between the first points of the raw trajectory and the straight line
    point_A_line = (x_interpolated[0], y_best_fit_line[0])
    x_difference = abs(trajectory_points[0][0] - point_A_line[0])
    y_difference = abs(trajectory_points[0][1] - point_A_line[1])

    # Check if the differences exceed the threshold, if so reverse the corresponding straight line coordinate
    if x_difference > similarity_threshold:
        x_interpolated = x_interpolated[::-1]  # Reverse the x-coordinates
    elif y_difference > similarity_threshold:
        y_best_fit_line = y_best_fit_line[::-1]  # Reverse the y-coordinates
```

```python
    # Export the straight-line points from A to B interpolated every 0.1 meters as a TXT file
    with open(os.path.join(parent_folder, f"straight_line_points-{csv_name}.txt"), "w") as f:
        for x, y in zip(x_interpolated, y_best_fit_line):
            f.write(f"({x}, {y})\n")

    # Create a spline representation of the smoothed and interpolated trajectory
    tck, u = splprep([x_interpolated, y_interpolated], s=0)
    spline_points = splev(np.linspace(0, 1, len(x_interpolated)), tck)

    # Determine the indices corresponding to every 5 meters along the spline
    step_size = 5
    spline_indices = np.arange(0, len(x_interpolated), step_size)

    # Create the simplified spline points
    simplified_spline_points = (spline_points[0][spline_indices], spline_points[1][spline_indices])


    # Initialize the plot
    fig, ax = plt.subplots(figsize=(16, 6))

    # Plot the interpolated trajectory, the reversed best-fit line, and the signed distances
    ax.scatter(x_interpolated, y_interpolated, c=point_colors, s=20, alpha=0.7, label="Interpolated Trajectory")
    ax.plot(x_interpolated, y_best_fit_line, "k-", linewidth=2, label="Best Fit Line")
    ax.quiver(x_interpolated, y_interpolated, np.zeros_like(x_interpolated), signed_distances, angles='xy', scale_units='xy', scale=1, width=0.002, color='blue', alpha=0.3)

    # Plot the simplified spline points
    ax.plot(*simplified_spline_points, "ro", markersize=8, label="Simplified Spline Points")

    # Initialize the plot
    fig, ax = plt.subplots(figsize=(16, 6))
    ax.plot(x_interpolated, y_best_fit_line, "k-", linewidth=2, label="Fixed Line")
    ax.axis("equal")
    ax.set_xlabel("X")
    ax.set_ylabel("Y")
    ax.set_title("Car Drift Behavior Near Fixed Line (Spline Distances)")


    padding = 10
    ax.set_xlim(min(x_interpolated) - padding, max(x_interpolated) + padding)
    ax.set_ylim(min(y_interpolated) - padding, max(y_interpolated) + padding)

    # Add car representation
    car_width = 2
    car_height = 1
    car = Rectangle((-car_width / 2, -car_height / 2), car_width, car_height, color="blue", angle=0)
    ax.add_patch(car)

    # Calculate the tangent angle along the simplified spline
    dx_simplified = np.gradient(simplified_spline_points[0])
    dy_simplified = np.gradient(simplified_spline_points[1])

    # Calculate the total number of frames for the animation
    total_frames = len(simplified_spline_points[0])

    # Create an array to store the trajectory colors
    trajectory_colors = np.zeros((len(spline_points[0]), 4))
    for i in range(len(spline_points[0])):
        trajectory_colors[i] = point_colors[i]

    # Pre-calculate the entire trajectory with the corresponding colors
    trajectory_segments = []
    for i in range(1, len(spline_points[0])):
        trajectory_segments.append(ax.plot(spline_points[0][i-1:i+1], spline_points[1][i-1:i+1], "-", color=trajectory_colors[i], linewidth=2)[0])

    # Hide the entire trajectory initially
    for segment in trajectory_segments:
        segment.set_visible(False)

    # Add text elements for stationing, signed distance, X, and Y values
    stationing_text = ax.text(0.02, 0.95, "", transform=ax.transAxes, fontsize=10, verticalalignment="top")
    signed_distance_text = ax.text(0.02, 0.90, "", transform=ax.transAxes, fontsize=10, verticalalignment="top")
    x_text = ax.text(0.02, 0.85, "", transform=ax.transAxes, fontsize=10, verticalalignment="top")
    y_text = ax.text(0.02, 0.80, "", transform=ax.transAxes, fontsize=10, verticalalignment="top")

    # Interpolate speed values
    speed_interpolator = interp1d(np.linspace(0, 1, len(speed_values)), speed_values, kind='cubic')
    speed_interpolated = speed_interpolator(np.linspace(0, 1, len(x_interpolated)))

    # Add speed values to the CSV export
    data["Speed"] = speed_interpolated
    data_df = pd.DataFrame(data)
    data_df.to_csv(os.path.join(parent_folder, f"speed-{csv_name}.csv"), index=False)

    # Add a text element for the speed value
    speed_text = ax.text(0.02, 0.75, "", transform=ax.transAxes, fontsize=10, verticalalignment="top")

    # Create an inset axes for the "camera"
    inset_ax = fig.add_axes([0.6, 0.6, 0.3, 0.3])
    inset_ax.set_aspect('equal')
    inset_ax.set_title("Camera View")

    # Set the zoom window size
    zoom_window_width = 7.5
    zoom_window_height = 7.5

    # Add the trajectory and the car to the inset axes
    inset_trajectory_segments = [inset_ax.plot(spline_points[0][i:i + 2], spline_points[1][i:i + 2],
                                               color=cmap(signed_distances[i]), linewidth=1.4, alpha=0.7)[0]
                                 for i in range(len(spline_points[0]) - 1)]

    # Add the straight line interpolated points to the inset axes
    inset_ax.plot(x_interpolated, y_best_fit_line, 'k:', linewidth=0.5, alpha=0.7)

    # Create the arrow for the inset axes
    inset_car = FancyArrowPatch((0, 0), (1, 1), mutation_scale=10, color='b', arrowstyle='-|>', linestyle='dashed')

    # Add the arrow to the inset axes
    inset_ax.add_patch(inset_car)

    # Add the straight line interpolated points to the inset axes
    inset_fixed_line = inset_ax.plot([], [], 'k-', linewidth=0.5, alpha=0.7)[0]

    # Update the fixed line within the zoom window
    inset_fixed_line.set_data(x_interpolated, y_best_fit_line)
    inset_fixed_line.set_color('black')  # Set the fixed line color to black

    # Update function for the animation
    def update(frame):
        end_idx = int(frame * len(spline_points[0]) / total_frames)

        # Show the trajectory segments up to the current frame
        for i in range(end_idx):
            trajectory_segments[i].set_visible(True)

        # Update car position and orientation
```

```python
            if frame < total_frames - 1:
                theta = np.arctan2(dy_simplified[frame], dx_simplified[frame])
                car.angle = np.degrees(theta)

                # Calculate the front center point of the car
                front_center_x = simplified_spline_points[0][frame] - car_width / 2 * np.cos(theta)
                front_center_y = simplified_spline_points[1][frame] - car_width / 2 * np.sin(theta)

                # Position the car so that the front center point aligns with the trajectory
                car.set_xy((front_center_x - car_width / 2, front_center_y - car_height / 2))

                # Update stationing, signed distance, X, and Y text elements
                stationing_text.set_text(f"Stationing: {stationing[end_idx]:.2f}")
                signed_distance_text.set_text(f"Signed Distance: {signed_distances[end_idx]:.2f}")
                x_text.set_text(f"X: {spline_points[0][end_idx]:.2f}")
                y_text.set_text(f"Y: {spline_points[1][end_idx]:.2f}")

                # Update the speed text
                speed_text.set_text(f"Speed: {speed_interpolated[end_idx]:.2f} km/h")

                # Update the camera and the zoomed window
                inset_ax.set_xlim(front_center_x - zoom_window_width / 2, front_center_x + zoom_window_width / 2)
                inset_ax.set_ylim(front_center_y - zoom_window_height / 2, front_center_y + zoom_window_height / 2)

                # Update the fixed line within the zoom window
                inset_fixed_line.set_data(x_interpolated, y_best_fit_line)

                # Show the trajectory segments up to the current frame in the inset axes
                for i in range(end_idx):
                    inset_trajectory_segments[i].set_visible(True)
                    inset_trajectory_segments[i].set_color(cmap(norm(np.abs(signed_distances[i]))))


                # Update the car position and orientation in the inset axes
                inset_car.xy = (front_center_x - car_width / 2, front_center_y - car_height / 2)
                inset_car._mutation_scale = 10
                inset_car.angle = car.angle
                inset_car.set_positions((front_center_x - car_width / 2, front_center_y - car_height / 2),
                                     (front_center_x + car_width / 2, front_center_y - car_height / 2))

                return car, stationing_text, signed_distance_text, x_text, y_text, speed_text, inset_car, *inset_trajectory_segments, inset_fixed_line

        # Set custom x and y intervals
        x_interval = 10
        y_interval = 10

        # Create grid
        ax.set_xticks(np.arange(min(x_interpolated) - padding, max(x_interpolated) + padding, x_interval))
        ax.set_yticks(np.arange(min(y_interpolated) - padding, max(y_interpolated) + padding, y_interval))
        ax.grid(color='lightgray', linestyle='-', linewidth=0.5)

        # Create the colorbar for the heatmap
        sm = plt.cm.ScalarMappable(cmap=cmap, norm=norm)
        sm.set_array([])
        cbar = plt.colorbar(sm, ax=ax)
        cbar.set_label('Signed Distance (m)')

        plt.rcParams['animation.embed_limit'] = 2**30  # 1 GB

        # Create the animation
        animation_duration = 50  # Animation duration in seconds
        ani = FuncAnimation(fig, update, frames=np.arange(0, total_frames), interval=1000 * animation_duration / total_frames)

        import matplotlib.animation as animation
        from moviepy.editor import VideoFileClip
        import os

        # Create a writer object
        Writer = animation.writers['ffmpeg']
        writer = Writer(fps=total_frames / animation_duration, metadata=dict(artist='Me'), bitrate=1800)



        # Define a unique name for each GIF and MP4 file
        gif_name = f"{csv_name}.gif"
        mp4_name = f"{csv_name}-TJ.mp4"
        converted_mp4_name = f"{csv_name}_converted-TJ.mp4"

        # Define the output paths
        gif_output_path = os.path.join(parent_folder, gif_name)
        mp4_output_path = os.path.join(parent_folder, mp4_name)
        converted_mp4_output_path = os.path.join(parent_folder, converted_mp4_name)

        # Save the animation as a GIF file
        ani.save(gif_output_path, writer="imagemagick", fps=total_frames / animation_duration)

        # Save the animation as a MP4 file
        ani.save(mp4_output_path, writer=writer)

        # Function to convert gifs to mp4s
        def convert_gifs_to_mp4s(directory, fps=24):
            # Walk through all files in the directory tree
            for root, dirs, files in os.walk(directory):
                for filename in files:
                    # Check if the file is a gif
                    if filename.lower().endswith(".gif"):
                        # Get the path of the gif
                        gif_path = os.path.join(root, filename)
                        # Get the path of the new mp4
                        mp4_path = os.path.join(root, filename[:-4] + "_converted.mp4")
                        # Load the gif
                        clip = VideoFileClip(gif_path)
                        # Resize the clip
                        clip_resized = clip.resize(width=1280)  # To resize the width to 1280 pixels
                        # Write the gif as an mp4
                        clip_resized.write_videofile(mp4_path, codec='libx264', fps=fps)

        # Use the function
        convert_gifs_to_mp4s(parent_folder)

if __name__ == "__main__":
    event_handler = MyHandler()
    observer = Observer()
    observer.schedule(event_handler, path=output_folder, recursive=True)  # `recursive=True` will watch all subdirectories
    observer.start()

    try:
        while True:  # keep the script running
            time.sleep(1)
    except KeyboardInterrupt:
        observer.stop()
    observer.join()
```

In [ ]: