Static frames

Name: Radu-Andrei Bourceanu

In this notebook we present an algorithm for identifying vehicles out of a 3 dimensional point cloud that contains carthesian coordonates gathered by a LiDAR.

Given the dataset, our task is to filter the point cloud for removing unnecessary data, applying the DBSCAN and K-Means algorithm for extracting the vehicles in the point cloud and comparing the two of them, and finally, draw clear bounding boxes around the vehicles and count them.

For the purpose of this project we use the following .csv files:

- `1_coche.csv` (1_car)
- `carretera.csv` (highway)
- `coche_coche.csv` (car_car)
- `coche_coche_moto.csv` (car_car_moto)

RAM cleanup

In [1]:
```
# Reset kernel => delete all variables
%reset

# OR

# Delete selected variables
'''
del df
import gc
gc.collect()
'''
```

Out[1]:
```
'\ndel df\nimport gc\ngc.collect()\n'
```

Imports

In [2]:
```
import numpy as np
import pandas as pd

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

from sklearn.cluster import DBSCAN
from sklearn.cluster import KMeans
from sklearn.neighbors import NearestNeighbors
from scipy.spatial import cKDTree

import math
```

Data dictionary

In [3]:
```
data = {
    '1': r'1_coche.csv',
    '2': r'coche_coche.csv',
    'moto': r'coche_coche_moto.csv',
    'empty': r'carretera.csv'
}
```

Select .csv file to work with (as defined in `data` above)

In [4]:
```
# Read csv file into df
df = pd.read_csv(data['2']) # Select here the desired .csv file
highway = pd.read_csv(data['empty']) # Don't change this

# Data conversion from float64 to float16
df = df.astype('float16')
highway = highway.astype('float16')

# Display dataframe
df
```

```
c:\Users\Radu Bourceanu\.conda\envs\adq_env\Lib\site-packages\pandas\core\dtypes\astype.py:133: RuntimeWarning: overflow encoun
tered in cast
  return arr.astype(dtype, copy=True)
c:\Users\Radu Bourceanu\.conda\envs\adq_env\Lib\site-packages\pandas\core\dtypes\astype.py:133: RuntimeWarning: overflow encoun
tered in cast
  return arr.astype(dtype, copy=True)
c:\Users\Radu Bourceanu\.conda\envs\adq_env\Lib\site-packages\pandas\io\formats\format.py:1458: RuntimeWarning: overflow encoun
tered in cast
  has_large_values = (abs_vals > 1e6).any()
```

Out[4]:

	x	y	z	intensity	t	reflectivity	ring	ambient	range
0	-0.000000	0.000000	0.000000	0.0	inf	0.0	0.0	0.0	0.0
1	-0.000000	0.000000	0.000000	16.0	0.0	0.0	0.0	1371.0	0.0
2	-0.000000	0.000000	0.000000	10.0	49888.0	0.0	0.0	990.0	0.0
3	-13.101562	2.755859	13.250000	12.0	inf	2.0	0.0	776.0	18816.0
4	-13.000000	2.816406	13.164062	14.0	inf	3.0	0.0	773.0	18688.0
...	...	...	...	...	...	...	...	...	...
131067	-0.000000	-0.000000	-0.000000	0.0	inf	0.0	127.0	0.0	0.0
131068	-0.000000	-0.000000	-0.000000	0.0	inf	0.0	127.0	0.0	0.0
131069	-0.000000	-0.000000	-0.000000	0.0	inf	0.0	127.0	0.0	0.0
131070	-0.000000	-0.000000	-0.000000	0.0	inf	0.0	127.0	0.0	0.0
131071	-0.000000	-0.000000	-0.000000	0.0	inf	0.0	127.0	0.0	0.0

131072 rows × 9 columns

Downsampling

- We downsample data for faster computation, as we don't need all the datapoints for an accurate representation of the vehicles in the point cloud
- We use Voxel Grid Downsampling, an industry standard

- Voxel Grid Downsampling works by dividing the space into multiple small cubes and calculating in each cube the mean position of all the datapoints that are within that space so that it afterwards replaces the datapoints with the one single obatined mean point
- A `voxel_size` of 0.1 reduces data by 80%

In [5]:
```python
df = df

def voxel_downsample(df, voxel_size=0.1):
    """
    Downsamples a point cloud for a single frame using a voxel grid.

    Args:
        df (pd.DataFrame): DataFrame with 'x', 'y', 'z' columns for one time frame.
        voxel_size (float): The side length of the voxel cube (e.g., 0.1 for 10cm).

    Returns:
        pd.DataFrame: The downsampled DataFrame.
    """

    # Create integer voxel indices by dividing coordinates by the voxel size
    voxel_indices = (df[['x', 'y', 'z']] / voxel_size).astype(int)

    # Keep only the first point encountered for each unique voxel index.
    # This is a highly efficient way to select one representative point per voxel.
    downsampled_df = df.loc[voxel_indices.drop_duplicates().index]

    return downsampled_df.reset_index(drop=True)

voxel_size = 0.05
print(f"Original DataFrame size: {len(df)} points")
print(f"Applying Voxel Grid Downsampling with voxel size {voxel_size}...")
df_downsampled = voxel_downsample(df, voxel_size=voxel_size)
print(f"Downsampled DataFrame size: {len(df_downsampled)} points")
print(f'Reduction of dataset size by {np.round(1-len(df_downsampled)/len(df), decimals=2)*100}%')

highway_downsampled = voxel_downsample(highway, voxel_size=voxel_size)
```

```
Original DataFrame size: 131072 points
Applying Voxel Grid Downsampling with voxel size 0.05...
Downsampled DataFrame size: 50879 points
Reduction of dataset size by 61.0%
```

Plot of the initial LiDAR point cloud

- In the following code we define a function for plotting the point cloud and do an initial plot of the point cloud

In [6]:
```
df = df_downsampled

def show_figure(df, title_text = 'LiDAR Point Cloud', color = 'Blue', zoom = 1):

    # Zoom factor of the figure as used in smartphone cameras
    zoom = 1/zoom

    # Create figure and scatter
    fig = go.Figure(
        data = [go.Scatter3d(x = df['x'], y = df['z'], z = -df['y'],
                             mode = 'markers', marker = dict(size = 1, colorscale = 'hot', opacity = 0.5, color = color))
                ],
        layout = go.Layout(scene = dict(xaxis = dict(title = 'X = X', range=[-40*zoom, 40*zoom]),
                                        yaxis = dict(title = 'Y = Z', range=[-40*zoom, 40*zoom]),
                                        zaxis = dict(title = 'Z = -Y', range=[-60*zoom, 20*zoom])),
                           width = 700,
                           height = 700,)
    )

    # Add title and
    fig.update_layout( title = dict(text = title_text, x = 0.5, y = 0.95, xanchor = 'center', yanchor = 'top',
                                    font = dict(family = "Arial, monospace", size = 35, color = "Gray", )),
    )

    # Show the figure
    fig.show()

show_figure(df, zoom=2.5)
```

Background Subtraction using Iterative Closest Point (ICP)

This script isolates moving objects (vehicles) from the 3D LiDAR point cloud by computationally "subtracting" a static background scan of an empty highway. The process involves three main stages:

1. Point Cloud Alignment: The Iterative Closest Point (ICP)
2. Optimized Nearest-Neighbor Search
3. Filtering by Distance

In [7]:
```python
df = df_downsampled

# --- Custom ICP Implementation ---
# Helper function
def best_fit_transform(A, B):
    '''
    Calculates the least-squares best-fit transform that maps corresponding points A to B in m spatial dimensions
    '''
    assert A.shape == B.shape
    m = A.shape[1]

    # get number of dimensions
    m = A.shape[1]

    # translate points to their centroids
    centroid_A = np.mean(A, axis=0)
    centroid_B = np.mean(B, axis=0)
    AA = A - centroid_A
    BB = B - centroid_B

    # rotation matrix
    H = np.dot(AA.T, BB)
    U, S, Vt = np.linalg.svd(H)
    R = np.dot(Vt.T, U.T)

    # special reflection case
    if np.linalg.det(R) < 0:
       Vt[m-1,:] *= -1
       R = np.dot(Vt.T, U.T)
```

```
    # translation
    t = centroid_B.T - np.dot(R,centroid_A.T)

    # homogeneous transformation
    T = np.identity(m+1)
    T[:m, :m] = R
    T[:m, m] = t

    return T, R, t

# ICP algorithm
def icp(A, B, max_iterations=20, tolerance=0.001):
    '''
    The Iterative Closest Point algorithm
    '''
    src = np.ones((4, A.shape[0]))
    dst = np.ones((4, B.shape[0]))
    src[:3,:] = np.copy(A.T)
    dst[:3,:] = np.copy(B.T)

    T = np.identity(4)

    neigh = NearestNeighbors(n_neighbors=1, algorithm='kd_tree')
    neigh.fit(B)

    prev_error = 0

    for i in range(max_iterations):
        distances, indices = neigh.kneighbors(src[:3,:].T, return_distance=True)
        T__ ,R ,t = best_fit_transform(src[:3,:].T, B[indices.ravel(),:])
        src = np.dot(T__, src)
        mean_error = np.mean(distances)
        if np.abs(prev_error - mean_error) < tolerance:
            break
        prev_error = mean_error

    T,R,t = best_fit_transform(A, src[:3,:].T)

    return T, distances, src[:3,:].T
```

```
# --- Main Script ---------------------------------------------------------------------------------------------------------
scene_df = df
background_df = highway_downsampled


# Convert to numpy arrays for processing
cols_to_use = ['x', 'y', 'z']
scene_points = scene_df[['x','y','z']].values
background_points = background_df[['x','y','z']].values

# 1. Apply the ICP algorithm for alignment
print(f"Running ICP alignment")
# We transform the background (highway) to match the scene (car)
T, _, _ = icp(background_points, scene_points)
print("Alignment complete.")

# 2.  Apply the calculated transformation to the background point cloud
# Convert full background_points to homogeneous coordinates (add a 4th dimension of 1s)
full_background_homogeneous = np.ones((4, len(background_points)))
full_background_homogeneous[:3, :] = background_points.T
# Apply the transformation
transformed_full_background_homogeneous = np.dot(T, full_background_homogeneous)
# Convert back to 3D coordinates
transformed_background_points = transformed_full_background_homogeneous[:3, :].T

# Create a DataFrame from the newly aligned FULL background points
aligned_background_df = pd.DataFrame(transformed_background_points, columns=['x', 'y', 'z'])

# 3. Subtract identical points to isolate distinct objects (Optimized)
print("Finding distinct points (Optimized background subtraction)...")
distance_threshold = 0.2# 0.15, 0.20cm threshold


# 4. Build a cKDTree for the aligned background points. This is a very fast C implementation.
background_tree = cKDTree(aligned_background_df.values)

# 5. Query the tree for nearest neighbors, but with a distance upper bound.
# This is the key optimization: the search is pruned for points that are far away.
distances, _ = background_tree.query(
    scene_df[['x', 'y', 'z']].values,
```

```
    k=1, # Find the single nearest neighbor
    distance_upper_bound=distance_threshold # Stop searching if no neighbor is found within this radius
)

# The query returns 'inf' (infinity) for points where no neighbor was found within the threshold.
# These are our distinct points.
is_distinct = distances == np.inf

# 6. Create the final dataframe with only the distinct points
distinct_objects_df = scene_df[is_distinct]

print(f"Original scene had \033[1m{len(scene_df)}\033[0m points.")
print(f"Found \033[1m{len(distinct_objects_df)}\033[0m distinct points belonging to new objects.")
print(f'Dataset reduction by {np.round(1-len(distinct_objects_df)/len(scene_df),decimals=2)*100}%')

show_figure(distinct_objects_df,title_text='Highway filtering with ICP', zoom=1)
```

```
Running ICP alignment
Alignment complete.
Finding distinct points (Optimized background subtraction)...
Original scene had 50879 points.
Found 2887 distinct points belonging to new objects.
Dataset reduction by 94.0%
```

Data clustering using DBSCAN

In the following codelines we will cluster the points to different categories using DBSCAN

In [8]:
```
df = distinct_objects_df

# Select only the first three columns
df = df.iloc[:, :3]

def apply_dbscan(df, eps=1.5, min_samples=50):
    # Only use rows without NaN in x, y, z
    xyz = df[['x', 'y', 'z']]
    dbscan = DBSCAN(eps=eps, min_samples=min_samples)
    labels = dbscan.fit_predict(xyz)
    # Map labels back to original dataframe index
    result = pd.Series(np.nan, index=df.index)
```

```
    result[xyz.index] = labels
    return result

color_dbscan = apply_dbscan(df)

# --- Create a Custom Color Palette for Clusters ---
# 1. Get the unique cluster labels found by HDBSCAN, excluding noise (-1)
unique_labels = sorted([label for label in color_dbscan.unique() if label != -1 and not pd.isna(label)])

# 2. Choose a dark, high-contrast color sequence from Plotly
color_sequence = px.colors.qualitative.Dark24

# 3. Create the color map dictionary
color_map = {
    # Assign a neutral color to noise points (label -1)
    -1: 'lightgrey',
    # Map each cluster label to a color from our chosen sequence
    **{label: color_sequence[i % len(color_sequence)] for i, label in enumerate(unique_labels)}
}

# 4. Create an array of colors for each point based on its cluster label
point_colors_db = color_dbscan.map(color_map).fillna('purple') # Use a fallback color for any potential NaNs

df['cluster'] = color_dbscan

df_dbscan = df

show_figure(df,title_text='Clustering using DBSCAN', color=point_colors_db, zoom=1)
```

```
C:\Users\Radu Bourceanu\AppData\Local\Temp\ipykernel_19808\3346859730.py:36: SettingWithCopyWarning:


A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
```

K-Means

Now with K-Means (**only for comparison purpose**, the algorithm is not suited for this task and delivers bad results)

In [9]:
```
df = distinct_objects_df

df = df.iloc[:, :3]

def apply_kmeans(df, n_clusters=2):
    """Applies K-Means clustering and returns the labels."""
    # Only use rows without NaN in x, y, z
    xyz = df[['x', 'y', 'z']].dropna()

    # Initialize and run the K-Means algorithm
    kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init='auto')
    labels = kmeans.fit_predict(xyz)

    # Map labels back to the original dataframe index to handle potential NaNs
    result = pd.Series(np.nan, index=df.index)
    result.loc[xyz.index] = labels
    return result

# Apply K-Means to the DataFrame, assuming we want to find 2 clusters
color_kmeans = apply_kmeans(df, n_clusters=4)

# --- Create a Custom Color Palette for Clusters ---
# 1. Get the unique cluster labels found by K-Means
unique_labels = sorted([label for label in color_kmeans.unique() if not pd.isna(label)])

# 2. Choose a dark, high-contrast color sequence from Plotly
color_sequence = px.colors.qualitative.Dark24

# 3. Create the color map dictionary
# Note: No need to handle the -1 noise label as K-Means doesn't have it
color_map = {
    label: color_sequence[i % len(color_sequence)] for i, label in enumerate(unique_labels)
}

# 4. Create an array of colors for each point based on its cluster label
point_colors_kmeans = color_kmeans.map(color_map).fillna('purple') # Fallback for any NaNs

# Create the final df_kmeans DataFrame with the cluster column
```

```
df['cluster'] = color_kmeans
df_kmeans = df

show_figure(df_kmeans, title_text='Clustering using K-Means', color=point_colors_kmeans, zoom=1)
```

```
C:\Users\Radu Bourceanu\AppData\Local\Temp\ipykernel_19808\1217981611.py:39: SettingWithCopyWarning:


A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
```

Bounding Boxes and final result

In [10]:
```
# Select which method to plot:
method = 'db'      #choose between 'db' for DBSCAN or 'km' for K-Means
if method == 'db':
    df = df_dbscan
    color = point_colors_db
else:
    df = df_kmeans
    color = point_colors_kmeans

#-------------------------------------------------------------------------------------
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
            centroid = cluster_points[['x', 'y', 'z']].mean().values

            # 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

#--------------------------------------------------------------------------------
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
#-----------------------------------------------------------------------------------------

objects = calculate_bounding_boxes(df)
for obj in objects:
    print(f"--- Object Cluster ID: {obj['cluster_id']} ---")
    print(f"  Centroid: {np.round(obj['centroid'], 2)}")
    print(f"  Min Corner (x,y,z): {np.round(obj['min_corner'], 2)}")
    print(f"  Max Corner (x,y,z): {np.round(obj['max_corner'], 2)}")
```

```
    print("-" * 30)

# Obtain box lines:
x_lines = []
y_lines = []
z_lines = []
for i in range(len(objects)):
    # Calculate distance from centroid to min_corner
    distance = math.sqrt((objects[i]['centroid'][0]-objects[i]['min_corner'][0])**2 +
                         (objects[i]['centroid'][1]-objects[i]['min_corner'][1])**2 +
                         (objects[i]['centroid'][2]-objects[i]['min_corner'][2])**2)
    # Set treshold for ignoring big boxes (noise not deleted from the highway)
    if distance < 5:
        box_lines = get_bounding_box_lines(objects[i]['min_corner'],objects[i]['max_corner'])
        x_lines.extend(box_lines[0])
        y_lines.extend(box_lines[2])    # Make to Z axis
        z_lines.extend(box_lines[1])    # Make to Y axis

# Make Z axis to -Y
z_lines = [-v if v is not None else None for v in z_lines]

# --- Create the Plot ----------------------------------------------------------------------------------------
# 1. Initialize a Figure object
zoom = 5
zoom = 1/zoom   # as defined in smartphone camera


fig = go.Figure(
    data = [
            go.Scatter3d(
                x = df['x'],
                y = df['z'],
                z = -df['y'],
                mode = 'markers',
                marker = dict(
                    size = 1, colorscale = 'hot', opacity = 0.5, color = color
                ),
            name = "LiDAR point cloud"
            )
        ],layout = go.Layout(
            scene = dict(
```

```
                xaxis = dict(title = 'X = X', range=[-60*zoom, 60*zoom]),
                yaxis = dict(title = 'Y = Z', range=[-60*zoom, 60*zoom]),
                zaxis = dict(title = 'Z = -Y', range=[-60*zoom, 60*zoom])
            ),
            width = 700,
            height = 700,
        ))

# 2. Add the bounding box lines as a 3D scatter plot in 'lines' mode
fig.add_trace(go.Scatter3d(
    x=x_lines,
    y=y_lines,
    z=z_lines,
    mode='lines',
    line=dict(
        color='red',  # Set the color of the box
        width=1       # Set the line width for better visibility
    ),
    name='Bounding Boxes'
))

# 2b. Add a label ("car") next to the bounding box
# Place the label at the centroid of the first object (or adjust as needed)
for i in range(len(objects)):
    distance = math.sqrt((objects[i]['centroid'][0]-objects[i]['min_corner'][0])**2 +
                         (objects[i]['centroid'][1]-objects[i]['min_corner'][1])**2 +
                         (objects[i]['centroid'][2]-objects[i]['min_corner'][2])**2)
    if distance < 5:
        centroid = objects[i]['min_corner']
        fig.add_trace(
            go.Scatter3d(
            x=[centroid[0]],
            y=[centroid[2]],   # Z axis in plot is Y in data
            z=[-centroid[1]],  # -Y axis in plot is Z in data
            mode='text',
            text=[f'Vehicle {i+1}'],
            textposition='top center',
            showlegend=False,
            textfont=dict(color='red', size=10)
            )
        )
```

```python
# 3. Update the layout for better viewing
fig.update_layout(
    title={
        'text': f"3D Bounding Box Visualization: {i+1} identified vehicles",
        'x': 0.5,  # Center the title
        'xanchor': 'center',
        'yanchor': 'top',
        'font': dict(
            family='Arial, sans-serif',  # Use a modern font
            size=28,                     # Increase font size
            color='black'
        )
    },
    scene=dict(
        xaxis_title='X Axis',
        yaxis_title='Y Axis',
        zaxis_title='Z Axis',
    ),
    width=700,
    height=700
)

fig.show()
```

```
--- Object Cluster ID: 0.0 ---
  Centroid: [-4.65  4.45  1.27]
  Min Corner (x,y,z): [-7.35  4.02  0.41]
  Max Corner (x,y,z): [-3.33  5.3   2.34]
------------------------------
--- Object Cluster ID: 1.0 ---
  Centroid: [-2.73  4.4  -1.98]
  Min Corner (x,y,z): [-5.54  3.88 -2.96]
  Max Corner (x,y,z): [-1.46  5.44 -1.36]
------------------------------
```