AER1217 Lab 3: Georeferencing Using UAV Payload Data

Yuhang Wang 1005048832
Andres Cervera Rozo 1005469217
Mohsen Diraneyya 1006462604

March 22, 2024

1 Introduction

In this lab, we used the Parrot AR Drone 2.0 bottom-facing camera to identify six target points on the ground and localize them using the Vicon inertial coordinate system to track the drone's location and orientation.

We followed a two-step approach for this estimation. First, we looped through the frames obtained from the camera and their associated drone state to identify and localize the target centroids in the inertial frame. This step can be run online as we obtain more frames and states. The second step involved sanitizing the locations of these centroids by using k-means clustering of all the target locations identified on all frames. This will guarantee the best possible estimation based on the frame estimation of these targets. Note that the k-means clustering can be implemented at the end of the flight or every k-frame. We ran it at the end of the flight to get the best possible estimate that includes all the frames.

The sections below explain the methods used during this process.

2 Image Processing

The goal of image processing is to identify the targets on the ground. For each frame image, we used OpenCV to read and undistorted the image. After the undistorted image, we changed the HSV cylindrical-coordinate colour scheme. This colour scheme makes it easier to distinguish different colours with brightness. Since we know the extent of this problem and the properties of the target, we exploited that in our target identification.

We applied two masks to the image. The first mask identifies regions of green that match the target colour, and the second mask identifies regions of white that match the target sticker colour. We then looked through all instances of green regions and sanitized them based on these three criteria:

- the area must be a contour from the green-masked image between HSV colour bounds of $(35, 70, 20)$ and $(80, 255, 255)$
- their area in pixels must be between 150 and 600 pixels
- their aspect ratio must be between 0.667 and 1.5
- they must be bounded by a contour from the white-masked image between HSV colour bounds of $(0, 0, 155)$ and $(255, 100, 255)$

After sanitizing the targets in the frame, we identify their centroid as the centroid of the green contour. Figure 1 shows two samples of the processed images using these two masks and Figure 2 shows a sample image with contour around the green dot and its white background.

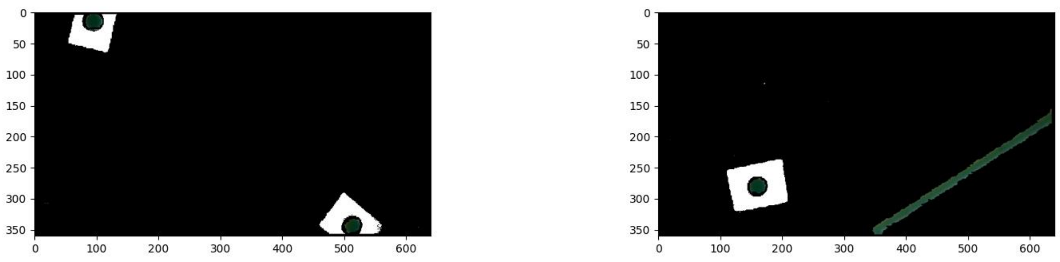


Figure 1: Sample Processed Frame Images

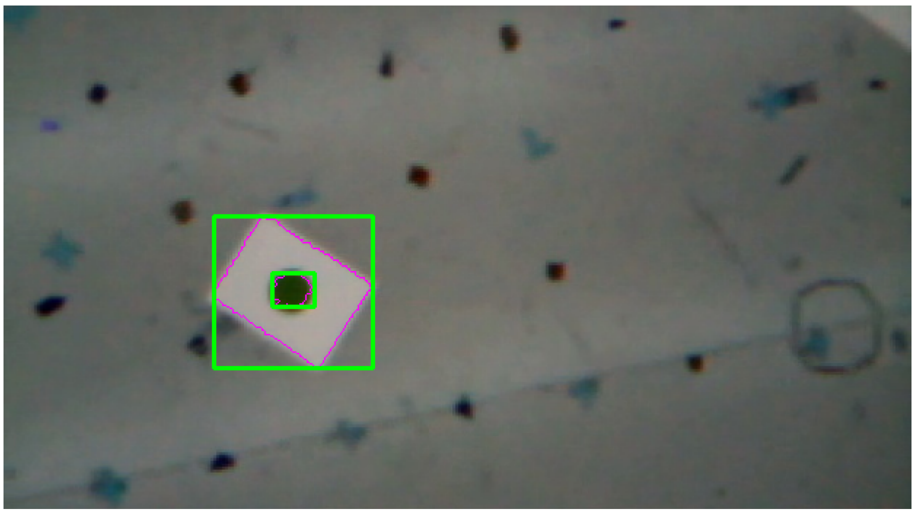


Figure 2: Undistorted image with bounding contours

3 Target Localization

The pose of the vehicle base is provided for all image frames through the Vicon motion capture system. Converting the provided quaternions into their respective rotation matrix made it possible to construct the transformation from the base to the world frame, T_{WC}. Additionally, the involutory transformation from the base frame to the camera frame is also provided as T_{CB} (equivalent to T_{BC}).

By exploiting the fact that these targets are known to be on the ground and have a world frame z coordinate of zero, it was possible to determine the distance in the z-axis of the vehicle frame to the ground. As a result, this z value can be used to complete the pinhole camera model and determine the coordinates of the green dots in the camera frame. These coordinates were then transformed to the inertial world frame through the previously mentioned transformation matrices to achieve the resulting inertial world frame x and y coordinates as shown in equations 1, 2 and 3.

First, we can solve the third equation $z_W = 0$ for z_C. Substituting this value on the right side, we can solve for x_W and y_W to get the inertial frame x and y coordinates.

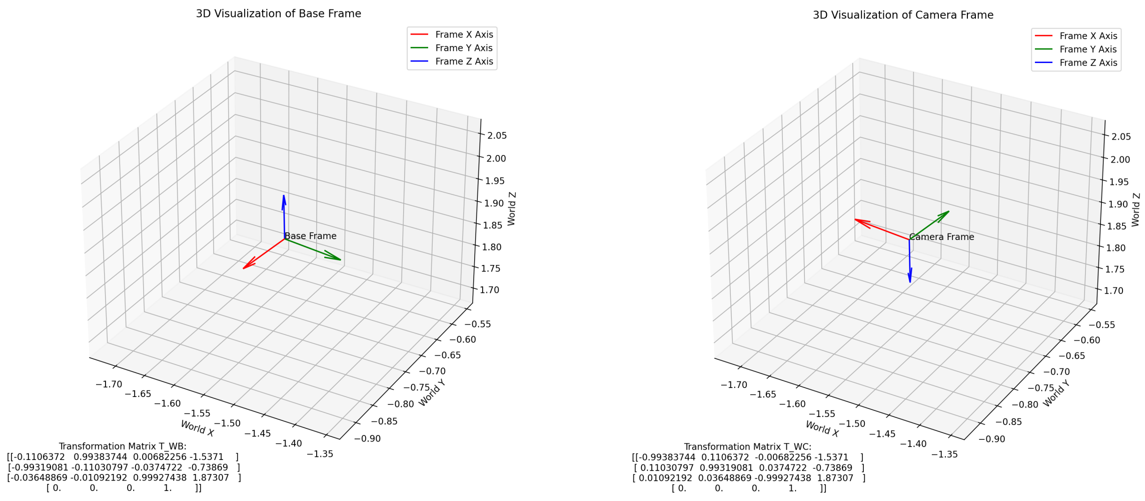
$$T_{WC} = T_{WB}T_{BC} \tag{1}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}_W = T_{WC} \begin{bmatrix} x \\ y \\ z \end{bmatrix}_C \tag{2}$$

$$\begin{bmatrix} x \\ y \\ 0 \end{bmatrix}_W = T_{WC} \begin{bmatrix} z * d_x/f_x \\ z * d_y/f_y \\ z \end{bmatrix}_C \tag{3}$$

$$\text{where } d_x \text{ and } d_y \text{ are the pixel x and y coordinates} \tag{4}$$

Figure 3 demonstrates a sample of an instance of the drone's vehicle base frame coordinate system and that of the camera frame. It can be seen that there is no translation between the 2 frames, and the rotation that is applied results in the z-axis pointing down, and the x and y axis are aligned with the conventional u and v axis of the camera image, respectively.



```
Transformation Matrix T_WB:
[[-0.1106372   0.99383744  0.00682256 -1.5371    ]
 [-0.99319081 -0.11030797 -0.0374722  -0.73869   ]
 [-0.03648869 -0.01092192  0.99927438  1.87307   ]
 [ 0.          0.          0.          1.        ]]
```

(a) Base Frame

```
Transformation Matrix T_WC:
[[-0.99383744  0.1106372  -0.00682256 -1.5371    ]
 [ 0.11030797  0.99319081  0.0374722  -0.73869   ]
 [ 0.01092192  0.03648869 -0.99927438  1.87307   ]
 [ 0.          0.          0.          1.        ]]
```

(b) Camera Frame

Figure 3: Sample Drone Coordinate Systems

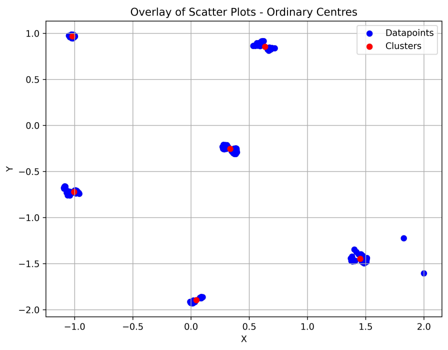
4 Filtered K-means Clustering

Once we get a list of data points, we implement a K-mean clustering algorithm to categorize them into clusters. This algorithm identifies and groups similar data points based on their distance. The algorithm iteratively updates the cluster center and reassigns points to the nearest cluster until convergence. Appendix A shows our code for the K-mean clustering algorithm.

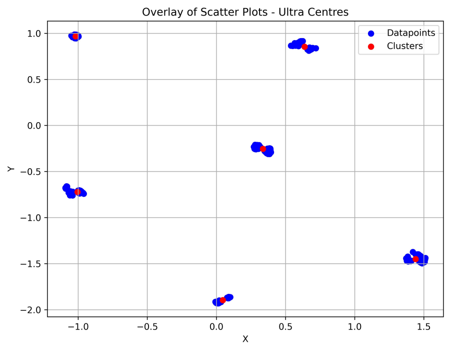
The process begins by initiating 6 cluster centers. These can be generated randomly, but in our case, they were assigned.

Then, we applied coordinate-clustering to obtain ordinary centers. Clustering was done by first assigning points to their cluster center, which is the center they are closest to. Then, the cluster center was recalculated by taking the mean of the data points that were assigned to them. These two steps were done iteratively 50 times, and the cluster center got closer to the ideal point after each iteration.

Figure 4a shows the data points and the cluster center. Note that some outliers are far away from any center points. Data points more than 0.3 meters away from any center point were removed. These cleaned data points were then clustered again to produce a cleaned cluster center. Data points then undergo another filtering process with a 0.1-meter threshold, and finally, a group of ultra-clean cluster centers were produced. This was done to remove any potential erroneous data that is far from the centroid centres and could potentially skew the estimate location if left unfiltered. Figure 4b shows the ultra-clean data points and ultra-clean cluster center.



(a) Ordinary Cluster Centre



(b) Ultra Clean Cluster Centre

Figure 4: Sample Drone Coordinate Systems

5 Results

The procedure described in the previous sections was applied to the data of the UAV flying above the 6 target green centres. After detecting the target in the image, localizing in the world frame, and clustering the resulting points, the algorithm produced 6 predictions for the locations of the 6 landmarks. These final cluster centres are detailed in Table 1 and are the corresponding numerical values to the red dots seen in Figure 4b. This algorithm can be applied both online and offline to iteratively improve on the accuracy of the landmark estimates.

X	**Y**
-1.016	0.969
0.056	-1.912
-1.006	-0.722
0.635	0.856
0.336	-0.253
1.441	-1.455

Table 1: Final Cluster Center

Appendix A: K-mean clustering

```
def get_distance(point1, point2):
    x1, y1 = point1
    x2, y2 = point2
    squared_diffs = (x2 - x1) ** 2 + (y2 - y1) ** 2
    distance = np.sqrt(squared_diffs)
    return distance


def assign_cluster(cluster_centres, valid_world_targets): # 6x2, ?x2
    cluster_ids = []
    for point in valid_world_targets:
        min_dist = float('inf')
        cluster_id = -1
        for c_id, cluster_point in enumerate(cluster_centres):
            distance = get_distance(point, cluster_point)
            if distance < min_dist:
                min_dist = distance
                cluster_id = c_id
        if cluster_id > -1:
            cluster_ids.append(cluster_id)
    cluster_ids = np.array(cluster_ids)
    return cluster_ids


def re_centre_clusters(cluster_ids, k_clusters, valid_world_targets, cluster_centres):
    for c_id in range(k_clusters):
        mask = (cluster_ids == c_id)
        members = valid_world_targets[mask]
        num_members = len(members)
        if num_members == 0:
            continue
        x_sum = 0
        y_sum = 0
        for point in members:
            x_sum += point[0]
            y_sum += point[1]
        x_centre = x_sum / num_members
        y_centre = y_sum / num_members
        cluster_centres[c_id] = [x_centre, y_centre]
    cluster_centres = np.array(cluster_centres)

    return cluster_centres


def filter_coordinates(cluster_centres, datapoints, tol):
    filtered_coords = []
    for coord in datapoints:
        for centre in cluster_centres:
            distance = get_distance(centre, coord)
            if distance <= tol:
                filtered_coords.append(coord)
                break
    clean_targets = np.array(filtered_coords)
    return clean_targets


def mini_k_means(k_clusters, datapoints, cluster_centres):
    cluster_ids = assign_cluster(cluster_centres, datapoints) # 1D array of length =
        len(datapoints)
    for _ in range(k_means_iterations):
        cluster_centres = re_centre_clusters(cluster_ids, k_clusters, datapoints,
            cluster_centres)
        cluster_ids = assign_cluster(cluster_centres, datapoints)
    estimated_final_centres = re_centre_clusters(cluster_ids, k_clusters, datapoints,
        cluster_centres)
    return estimated_final_centres


def plot_k_means_results(datapoints, clusters, label='targets'):
    plt.figure(figsize=(8, 6))
    plt.scatter(datapoints[:, 0], datapoints[:, 1], color='blue', label='Datapoints')
    plt.scatter(clusters[:, 0], clusters[:, 1], color='red', label='Clusters')
    plt.title('Overlay of Scatter Plots - ' + label)
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.legend()
    plt.grid(True)
    plt.show()


def apply_full_k_means(valid_world_targets):
    k_clusters = 6
    cluster_centres = np.array([
        [-1.0, 1.0],
        [0.0, -1.0],
        [-1.0, -1.0],
        [1.0, 1.0],
        [1.0, 0.0],
        [1.5, -1.0]
    ])
    ordinary_centres = mini_k_means(k_clusters, valid_world_targets, cluster_centres)
    plot_k_means_results(valid_world_targets, ordinary_centres, 'Ordinary Centres')

    clean_targets = filter_coordinates(ordinary_centres, valid_world_targets,
        k_means_tolerance)
    clean_centres = mini_k_means(k_clusters, clean_targets, ordinary_centres)
    plot_k_means_results(clean_targets, clean_centres, 'Clean Centres')

    ultra_clean_targets = filter_coordinates(clean_centres, clean_targets, 0.1)
    ultra_clean_centres = mini_k_means(k_clusters, ultra_clean_targets, clean_centres)
    plot_k_means_results(ultra_clean_targets, ultra_clean_centres, 'Ultra Centres')

    return ultra_clean_centres
```

Appendix B: Image Processing and Target Localization

```
import numpy as np
import cv2
import pandas as pd
import os
import matplotlib.pyplot as plt
from scipy.spatial.transform import Rotation
# Parameters
k_means_iterations = 50
k_means_tolerance = 0.3
min_area = 150
max_area = 600 # exclude green tape
min_aspect_ratio = 0.6667
max_aspect_ratio = 1.5
np.random.seed(1217)
lower_green = np.array([35, 70, 20])
```

```python
upper_green = np.array([80, 255, 255])
white_sensitivity = 100
lower_white = np.array([0, 0, 255 - white_sensitivity])
upper_white = np.array([255, white_sensitivity, 255])
current_directory = os.getcwd()
# Input files locations
file_name = 'lab3_pose.csv'
image_dir = 'image_folder/input_frames'
out_image_dir = 'image_folder/processed_frames'
save_out_images = False
# Get Images
file_path = os.path.join(current_directory, file_name)
image_dir = os.path.join(current_directory, image_dir)
image_frames = sorted([int(i[6:-4]) for i in os.listdir(image_dir)])
image_jpgs = [f'{image_dir}/image_{j}.jpg' for j in image_frames]


def visualize_transform(t_matrix):
    buffer = 0.2
    # Extract rotation matrix and translation vector from the transformation matrix
    r_matrix = t_matrix[:3, :3]
    d_to = t_matrix[:3, 3]

    # Create a new figure
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.quiver(d_to[0], d_to[1], d_to[2], r_matrix[0, 0], r_matrix[1, 0], r_matrix[2, 0],
        color='r', length=0.1, arrow_length_ratio=0.3)
    ax.quiver(d_to[0], d_to[1], d_to[2], r_matrix[0, 1], r_matrix[1, 1], r_matrix[2, 1],
        color='g', length=0.1, arrow_length_ratio=0.3)
    ax.quiver(d_to[0], d_to[1], d_to[2], r_matrix[0, 2], r_matrix[1, 2], r_matrix[2, 2],
        color='b', length=0.1, arrow_length_ratio=0.3)
    ax.text(d_to[0], d_to[1], d_to[2], 'Transform', color='k')

    # Set plot limits and labels
    ax.set_xlim([d_to[0] - buffer, d_to[0] + buffer])
    ax.set_ylim([d_to[1] - buffer, d_to[1] + buffer])
    ax.set_zlim([d_to[2] - buffer, d_to[2] + buffer])
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    # Show plot
    plt.show()


# ---------------- DEPTH ---------------- #
def get_frame_state(frame_state):
    vehicle_translations = np.array([frame_state.p_x, frame_state.p_y, frame_state.p_z])
    vehicle_quaternion = np.array([frame_state.q_x, frame_state.q_y, frame_state.q_z,
        frame_state.q_w])
    r_matrix_wb = Rotation.from_quat(vehicle_quaternion).as_matrix() # correct,
        orthogonal, det(R)==1 SO group
    t_matrix_wb = np.eye(4) # Identity matrix
    t_matrix_wb[:3, :3] = r_matrix_wb # Set rotation values
    t_matrix_wb[:3, 3] = vehicle_translations
    return t_matrix_wb # would be depth, but cant trust rotation information


# ---------------- IMAGE PROCESSING ---------------- #
def realtime_frame_target_location(image_path, camera_calib, camera_distortion,
     t_matrix_wb, t_matrix_cb):
    image = cv2.imread(image_path)
```

```python
if image is None:
    print("No image")
    return None

undistorted_image = cv2.undistort(image, camera_calib, camera_distortion) # Get
     corrected image
hsv = cv2.cvtColor(undistorted_image, cv2.COLOR_BGR2HSV)
# Define lower and upper bounds for green color in HSV

masked_image = cv2.inRange(hsv, lower_green, upper_green)
white_masked_image = cv2.inRange(hsv, lower_white, upper_white)

# Find contours in the masked_image
contours, _ = cv2.findContours(masked_image, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
white_contours, _ = cv2.findContours(white_masked_image, cv2.RETR_EXTERNAL,
     cv2.CHAIN_APPROX_NONE)
target_world_coordinates_list = np.array([])

# Check if any contours are found
if len(contours) > 0:
    valid_cnt = []
    valid_rect = []
    valid_white_rect = []
    for cnt in contours:
        if min_area < cv2.contourArea(cnt) < max_area:
            rect = np.int16(cv2.boundingRect(cnt))
            aspect_ratio = float(rect[2]) / rect[3]

            if min_aspect_ratio < aspect_ratio < max_aspect_ratio:
                for white_cnt in white_contours:
                    white_rect = np.int16(cv2.boundingRect(white_cnt))
                    if (white_rect[0] < rect[0]) and ((white_rect[0] + white_rect[2]) >
                         (rect[0] + rect[2])) and (white_rect[1] < rect[1]) and (
                            (white_rect[1] + white_rect[3]) > (rect[1] + rect[3])):
                        valid_cnt.append(cnt)
                        valid_rect.append(rect)
                        valid_white_rect.append(white_rect)
                        break

    for i in range(len(valid_cnt)):
        this_rect = valid_rect[i]
        this_white_rect = valid_white_rect[i]
        centroid_x = this_rect[0] + this_rect[2] / 2
        centroid_y = this_rect[1] + this_rect[3] / 2

        c_x = camera_calib[0][2]
        c_y = camera_calib[1][2]
        f_x = camera_calib[0][0]
        f_y = camera_calib[1][1]

        delta_x = centroid_x - c_x
        delta_y = centroid_y - c_y
        t_matrix_bc = t_matrix_cb # unique matrix that happens to be its own inverse
        # find dots world coordinates
        t_matrix_wc = np.dot(t_matrix_wb, t_matrix_bc)
        camera_z = (0 - t_matrix_wc[2, 3]) / (t_matrix_wc[2, 0] * delta_x / f_x +
             t_matrix_wc[2, 1] * delta_y / f_y + t_matrix_wc[2, 2])
        camera_x = camera_z * delta_x / f_x
        camera_y = camera_z * delta_y / f_y

        target_cam_coordinates = np.array([[camera_x], [camera_y], [camera_z], [1]])
        target_world_coordinates = np.dot(t_matrix_wc, target_cam_coordinates)
        target_world_coordinates = np.array(within_bounds(target_world_coordinates))
```

```
            if len(target_world_coordinates_list) > 0:
                target_world_coordinates_list = np.vstack((target_world_coordinates_list,
                     target_world_coordinates))
            else:
                target_world_coordinates_list = target_world_coordinates
            if save_out_images:
                output_img = undistorted_image.copy()
                output_img[np.where(masked_image == 0)] = 0
                output_img[np.where(white_masked_image != 0)] = 255
                plt.imshow(output_img)
                plt.savefig(os.path.join(os.path.join(current_directory, out_image_dir),
                f"image_{int(image_path.split('image_')[-1][:-4])}
                    _target_{target_world_coordinates_list.shape[0]}.jpg"))

    return target_world_coordinates_list # return target_cam_coordinates #4x1 [[x], [y],
        [z], 1]


# ---------------- TRANSFORMS ---------------- #
def within_bounds(target_world_coordinates):
    x = target_world_coordinates[0][0]
    y = target_world_coordinates[1][0]
    x_constrained = min(max(x, -2), 2)
    y_constrained = min(max(y, -2), 2)
    return np.array([[x_constrained, y_constrained]])


# ---------------- MAIN ---------------- #
def main():
    camera_intrinsic_mat = np.array([ # Camera Intrinsic Matrix
        [730.86, 0.0, 306.91],
        [0.0, 699.13, 150.34],
        [0.0, 0.0, 1.0]
    ])
    camera_distortion = np.array([0.191887, -0.563680, -0.003676, -0.002037, 0.0]) #
        Distortion Coefficients
    t_matrix_cb = np.array([ # extrinsic transformation matrix from the vehicle body
        (Vicon) frame to the camera frame
        [0.0, -1.0, 0.0, 0.0],
        [-1.0, 0.0, 0.0, 0.0],
        [0.0, 0.0, -1.0, 0.0],
        [0.0, 0.0, 0.0, 1.0]
    ])

    drone_state = pd.read_csv(file_path) #Vicon inertial vehicle information x in [-2.0,
        2.0] m and y in [-2.0, 2.0] m
    no_valid_images_found = True
    valid_world_targets = []

    # Real-time frame target coordinates
    for i in range(1, len(image_frames)):
        frame = image_frames[i]
        image_file_path = image_jpgs[i]

        if frame not in drone_state.index:
            raise ValueError("Index not found in DataFrame")

        t_matrix_wb = get_frame_state(drone_state.loc[frame - 1]) # there seems to be a
            lag of 1 frame
        target_world_coordinates = realtime_frame_target_location(image_file_path,
            camera_intrinsic_mat, camera_distortion, t_matrix_wb, t_matrix_cb) # return
            4x1
        if len(target_world_coordinates) > 0:
            print(f"image #: {frame} - Valid")

            if no_valid_images_found:
                valid_world_targets = target_world_coordinates
                no_valid_images_found = False
            else:
                valid_world_targets = np.vstack((valid_world_targets,
                    target_world_coordinates))

        else:
            print(f"image #: {frame} - Invalid")

    # Sanitize using k-means clustering
    valid_world_targets = np.array(valid_world_targets)
    final_clusters_prediction = apply_full_k_means(valid_world_targets)
    print("6 targets estimated at: ")
    print(final_clusters_prediction)
    return final_clusters_prediction


if __name__ == '__main__':
    target_estimates = main()
```