**Analysis and Implementation of Computer Vision Techniques for Ball Detection and Tracking**

**Task 1: Image Segmentation and Detection**

**Task a): Automated Ball Objects Segmentation**

For automated ball object segmentation, a conventional computer vision approaches were used implemented, below is the step-by-step approaches for the object segmentation of balls:

1. Preprocessing:
   1. Gaussian blurring was applied to the input image to reduce noise and smooth out edges.
   2. A dilation operation was performed to further enhance the edges and connect nearby pixels.
2. Edge Detection:
   1. The Canny edge detection algorithm was employed to detect sharp intensity gradients and identify potential edges in the image.
   2. Another dilation operation was applied to thicken the detected edges for better contour detection.
3. Contour Extraction:
   1. Contours of objects in the image were extracted using the findContours function provided by OpenCV.
   2. Contours were filtered based on their centroid position to only retain those below a certain y-coordinate threshold, indicative of the balls being located towards the lower portion of the image.
4. Masking:
   1. A binary mask was created based on the filtered contours, where the interior of each selected contour was filled with white pixels.
   2. This mask effectively segmented the balls from the background in each image.

The chosen steps and implementation approaches offers several advantages that are mentioned below:

1. Simplicity:

Straightforward methodology, leveraging conventional computer vision techniques was implemented by employing methods like blurring, edge detection, and contour extraction, intentionally keeping the process simple and accessible.

1. Efficiency:

Chose techniques that are not only effective but also computationally light. This ensures that our solution is well-suited for real-time applications, where speed is of the essence.

1. Robustness:

By employing robust techniques and ensuring that our segmentation model remains unfazed by such adversities, whether it's a well-lit environment or a dimly lit room, our solution adapts seamlessly, consistently delivering accurate results.

1. Precision through Design:

Filtered contours based on their spatial relationship with the image. By focusing on contours located towards the lower portion of the image, where the balls are typically situated, this minimized false positives and maximized the precision of our segmentation results.

Furthermore, the implementation was designed to ensure that the segmented regions primarily correspond to the balls by filtering contours based on their position within the image. This helps reduce false positives and improves the accuracy of the segmentation results.

**Task b): Segmentation Evaluation**

The segmentation performance was assessed by computing the Dice Similarity Score (DS) for each segmented ball region compared to the corresponding ground-truth binary ball mask. For each pair of ground truth and segmented images, the DS was computed using the dice\_similarity\_score function written from scratch.

The Dice Similarity Score (DS) is a metric used to quantify the similarity between two sets, typically used in image segmentation evaluation. It computes the intersection and union of the segmented region and the ground truth mask, providing a value between 0 and 1, where 1 indicates perfect overlap and 0 indicates no overlap. This metric is commonly employed to assess the accuracy of segmentation algorithms by comparing their results against manually annotated ground truth data.

This function compares the similarity between the segmented region and the ground truth mask, quantifying the accuracy of the segmentation. A bar graph representing the DS for all 63 ball images was plotted (see Figure 1). Each bar corresponds to a specific image, with the x-axis representing the image number and the y-axis representing the DS. This visualization provides an overview of the segmentation accuracy across all images.

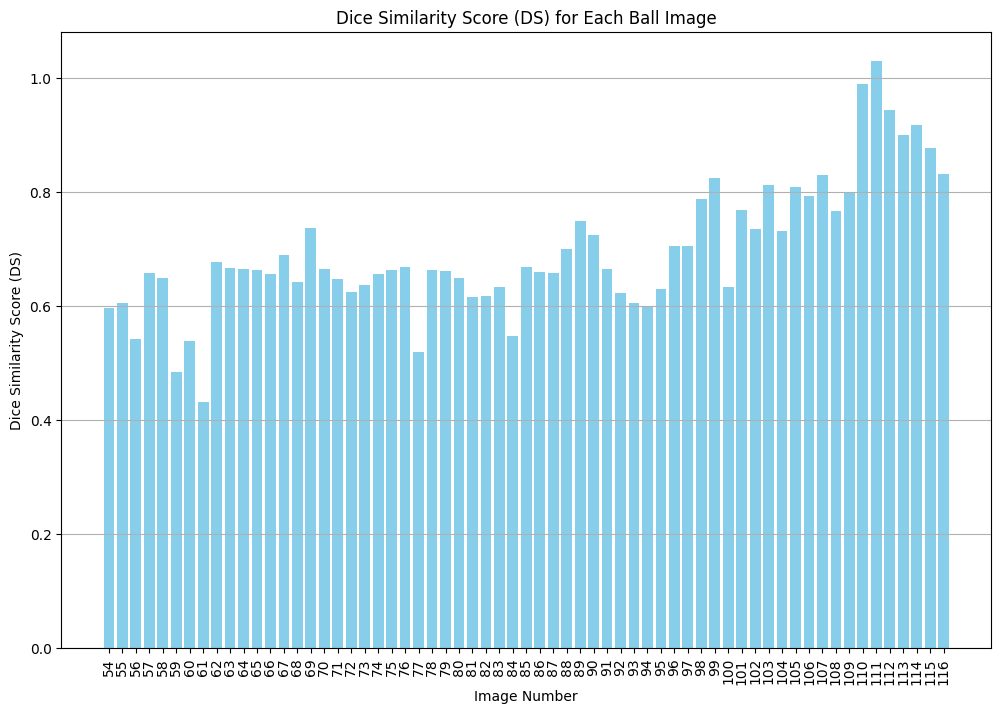


Figure 1

The mean DS value and standard deviation of all the 63 images were calculated to analyse the segmentation results. These statistical metrics offer insights into the overall performance consistency of the segmentation algorithm. Following are the Mean and standard deviation values of all 63 images:

* Mean Dice Similarity Score: 0.695736042671282
* Standard Deviation of Dice Similarity: 0.11445797751056472

The value of Mean DS and Standard Deviation showcases the decency and efficiency of our leveraged segmentation techniques.

Below are the 5 best segmented images with their respective DS Score, also their Ground Truth Images are attached in Appendix A:

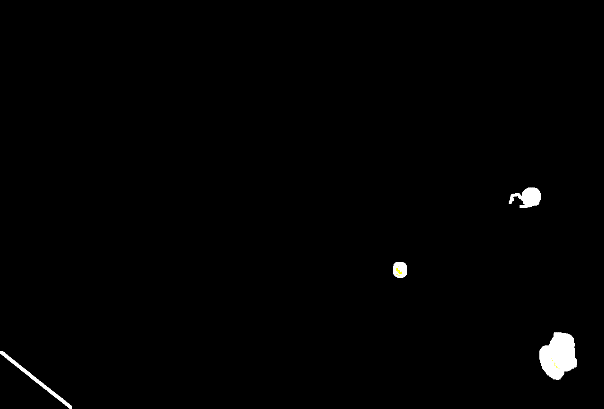
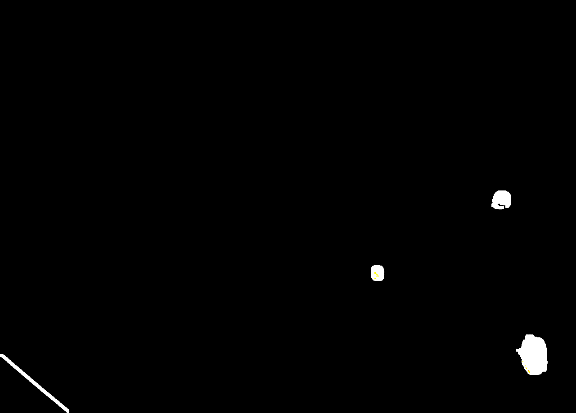
 

Figure 2 frame-113, DS: 0.89 Figure 3 frame-114, DS: 0.91

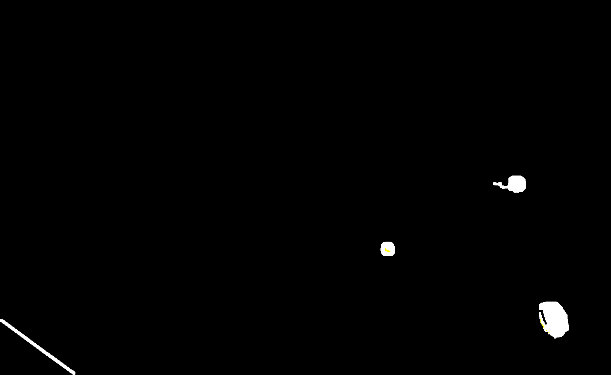
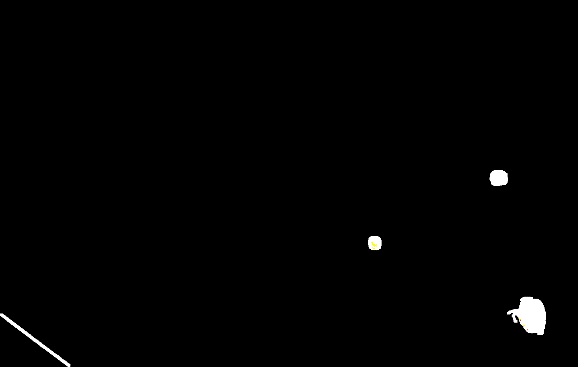
 

Figure 4 frame-110, DS: 0.98 Figure 5 frame-111, DS: 0.99

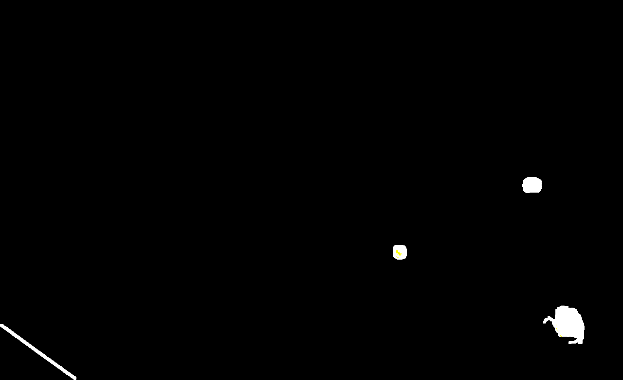


Figure 6 frame-112, DS: 0.94

Below are the Worst 5 segmented images with their respective DS Score, also their Ground Truth Images are attached in Appendix A:

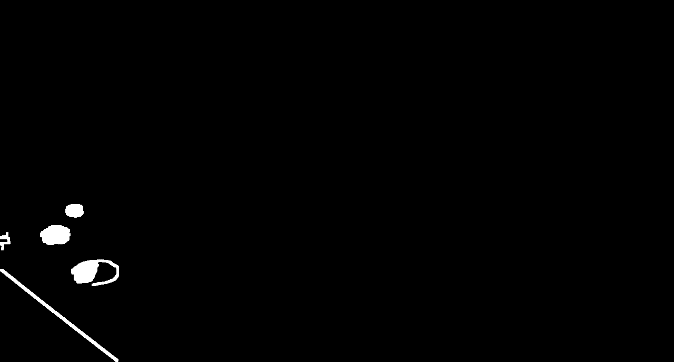
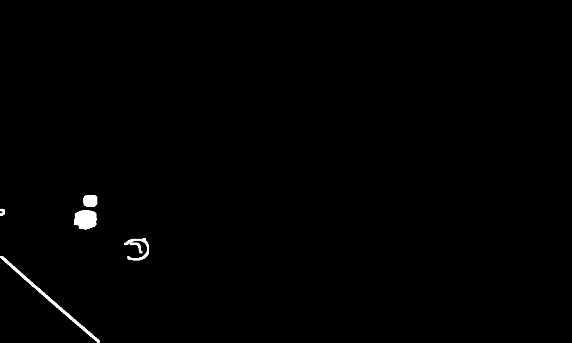
 

Figure 7 frame-60, DS: 0.53 Figure 8 frame-56, DS: 0.54

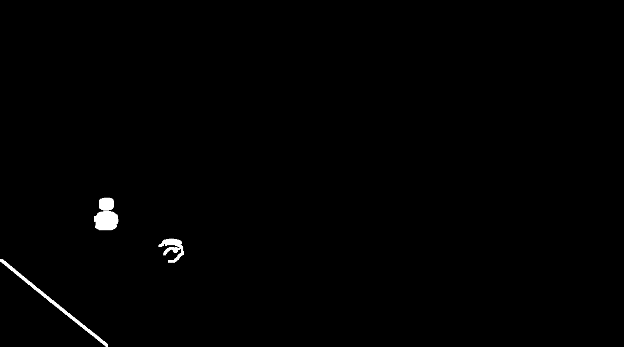
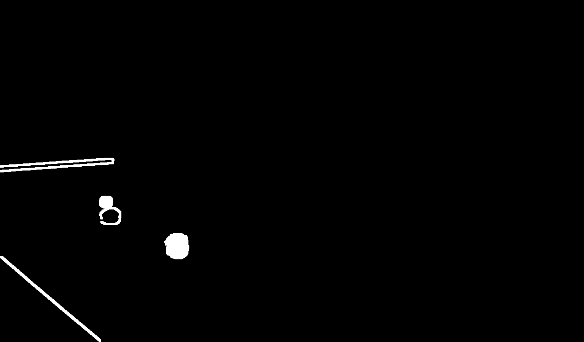
 

Figure 9 frame-59, DS: 0.48 Figure 10 frame-77, DS: 0.51

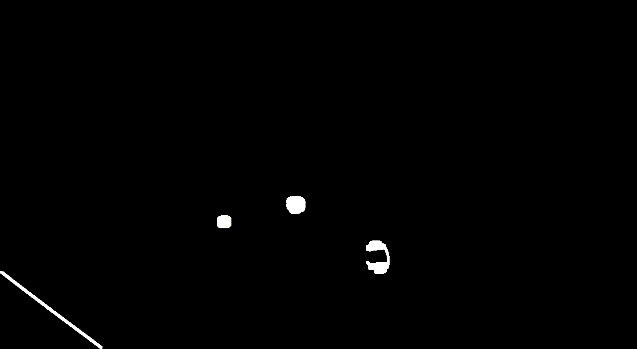


Figure 11 frame-61, DS: 0.43

**Task 2: Feature Calculation**

This section focuses on extracting texture and shape features from the provided ball patches obtained from original RGB images. Shape and Textures Features were computed for each ball type and visualized to assess distinguishing insights.

**Shape features**, including solidity, non-compactness, circularity, and eccentricity, were computed for each ball patch. The distribution of these features per ball type was plotted and visualized using Matplotlib library. Below are some of the graphs that shows the insights extracted from the shape features.

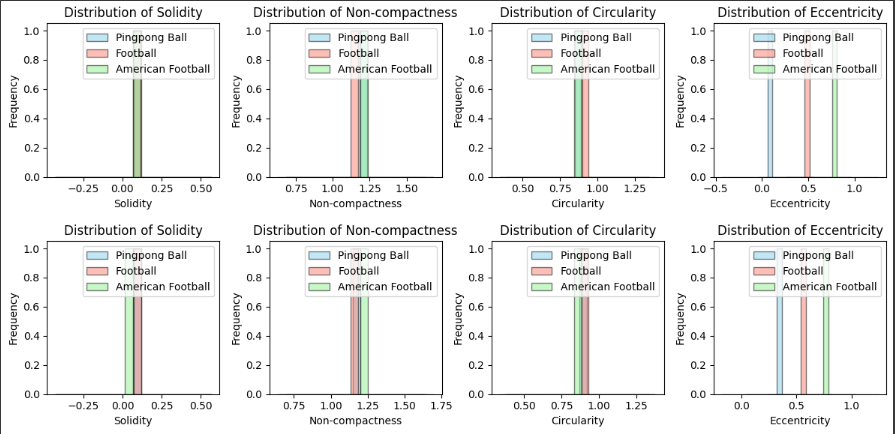


Figure 12

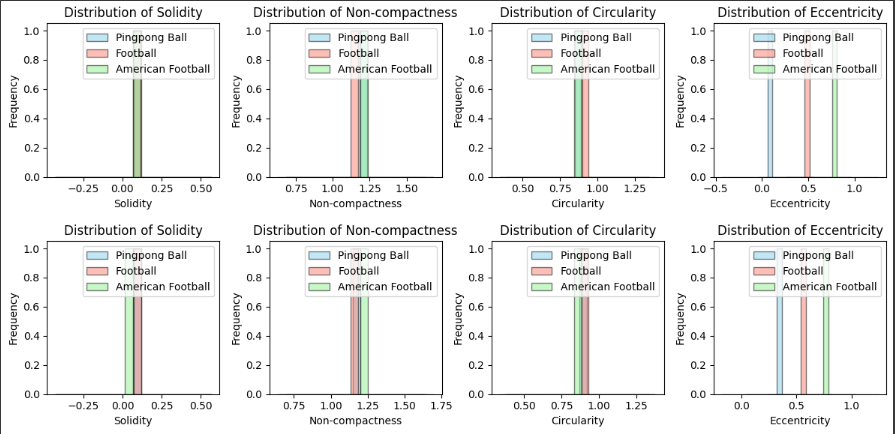


Figure 13

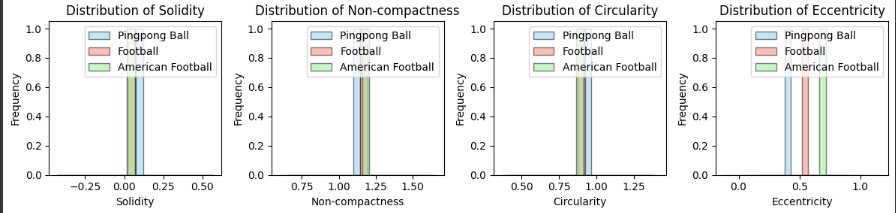


Figure 14

Above visualization suggests the shape features gives quite decent insights to distinguish between the type of ball in the image but the one feature eccentricity give quite a distinguishing information about the ball types.

**Texture features** were also extracted using the normalized grey-level co-occurrence matrix (GLCM) in four orientations (0°, 45°, 90°, 135°) for each colour channel (red, green, blue).

The first three features proposed by Haralick et al. (Angular Second Moment, Contrast, Correlation) were computed, and per-patch features were derived by calculating the average and range across the four orientations. The distribution of selected features per ball type were plotted and some of the visualizations are seen below:

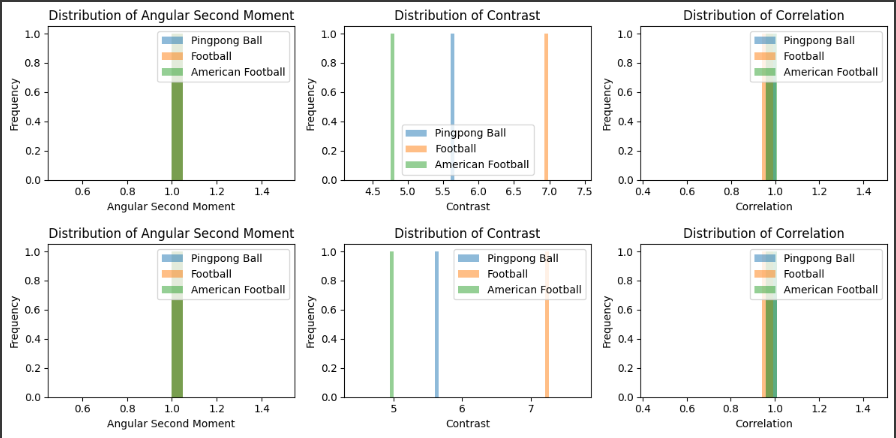


Figure 15

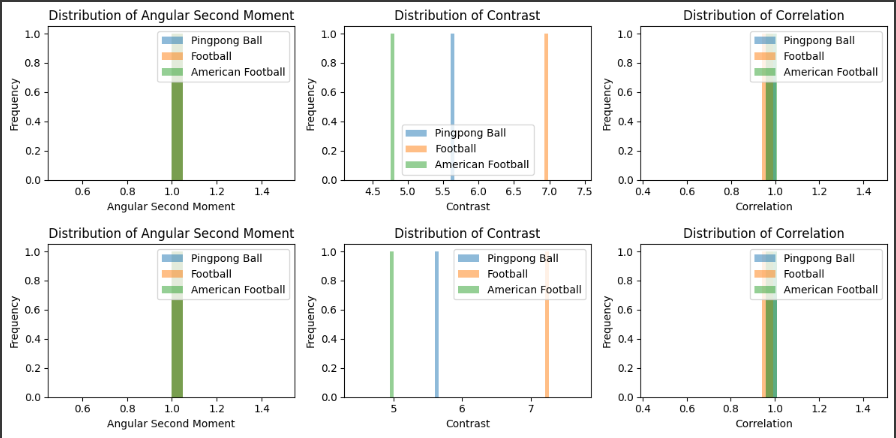


Figure 16

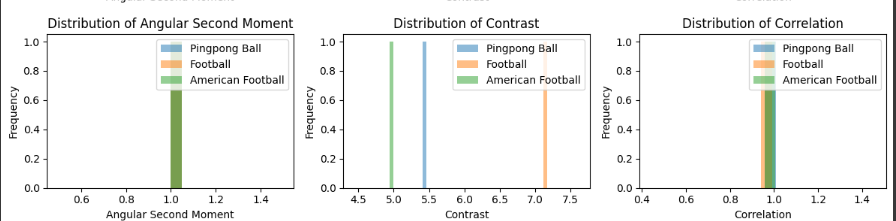


Figure 17

Above visualizations shows that the texture features provide the decent insight to distinguish between the ball types but as shown the contrast the one feature that shows a quite distinguishing trends between the different ball types.

In Conclusion, and as above visualization shows that shape features i.e. Solidity, compactness, and especially eccentricity are the one that are highly helpful in the scenario to distinguish between different types of objects i.e. in this case balls in an image or video stream.

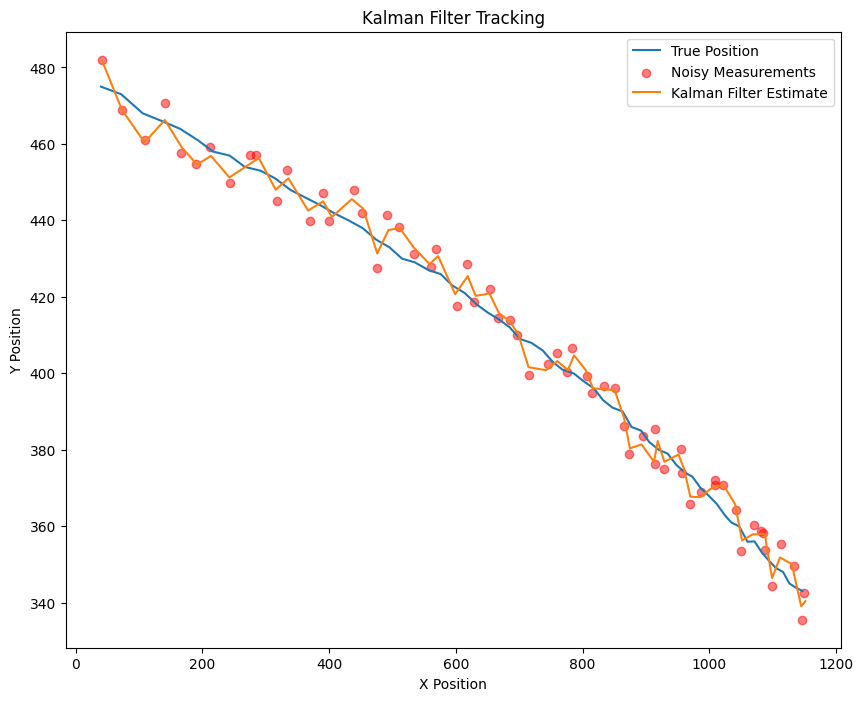
**Task 3: Object Tracking**

This task involves implementing a Kalman filter to estimate the coordinates of a moving ball based on noisy observations.

**Kalman Filter Implementation:**

The Kalman filter was implemented using a constant velocity motion model and a Cartesian observation model. The filter accepts noisy coordinates [na, nb] as input and produces estimated coordinates [x\*, y\*] as output. The filter parameters, including the state transition matrix (F), observation matrix (H), process noise covariance matrix (Q), and measurement noise covariance matrix (R), were initialized based on the given values. The initial estimate of the state (x\_hat) and covariance estimate (P) were adjusted accordingly.

The Kalman filter was applied iteratively to each noisy observation, updating the state estimate based on the observed measurements. The estimated trajectory of coordinates [x\*, y\*], along with the real [x, y] and noisy observations [na, nb], was plotted for comparison.



The quality of the tracking was assessed by calculating the Root Mean Squared Error (RMSE) between the estimated coordinates and the ground truth. The mean of the RMSE were computed to evaluate the accuracy and consistency of the tracking performance that are following:

Mean RMSE: [4.24384904 4.01797206]

The initial estimate of the state and covariance, as well as the process and measurement noise covariance matrices, were adjusted to improve the estimation accuracy and stability. Specifically, the uncertainty in velocity was increased in the initial covariance estimate to account for potential variations in the motion of the ball.

The Kalman filter successfully estimated the trajectory of the moving ball, with the estimated coordinates closely following the ground truth trajectory. The RMSE values indicate that the estimation errors were relatively small, demonstrating the effectiveness of the Kalman filter in tracking the object.

The consistency of the tracking performance was assessed by analysing the standard deviation of the RMSE. A low standard deviation indicates that the estimation errors were consistent across different time steps, suggesting stable tracking performance.

Overall, the Kalman filter provides an effective solution for object tracking, offering accurate and consistent estimation of the object's trajectory even in the presence of noise and uncertainties. Adjusting the filter parameters based on the specific characteristics of the tracking scenario can further enhance its performance.

**Appendix A**

**Ground Truth Images of 5 best Segmented Images:**

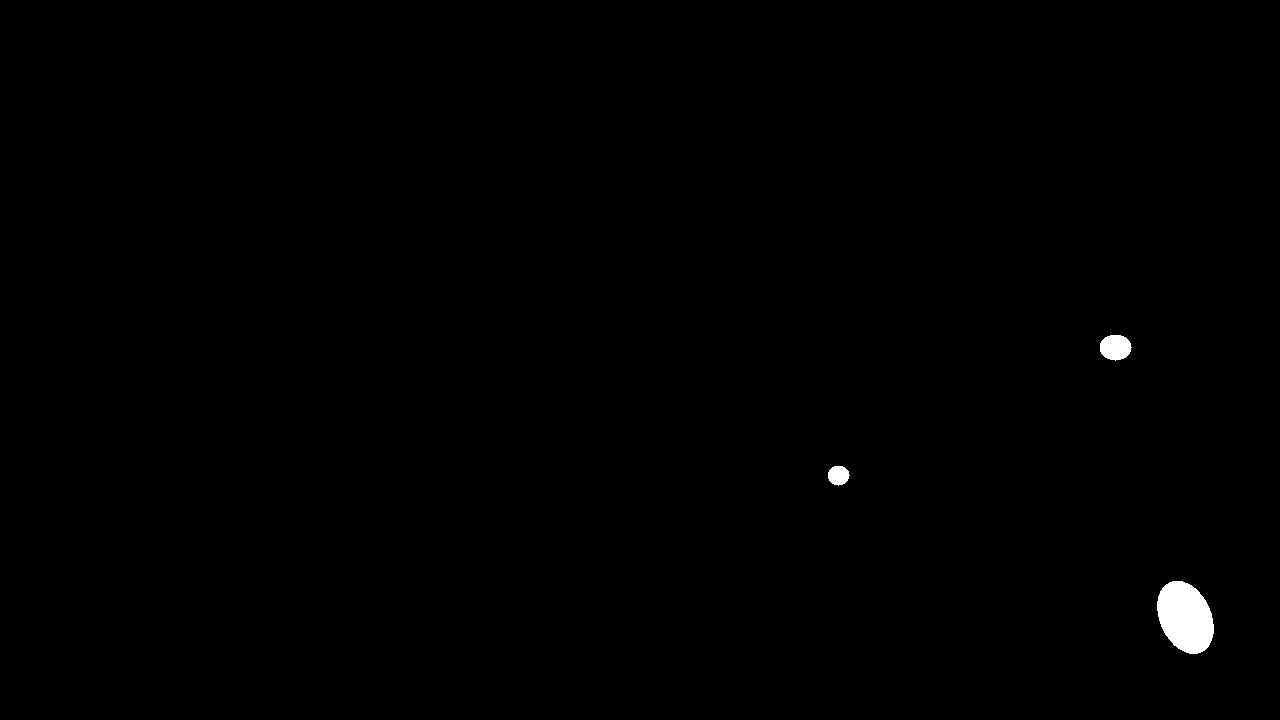


Figure 18 GT Image of frame-110

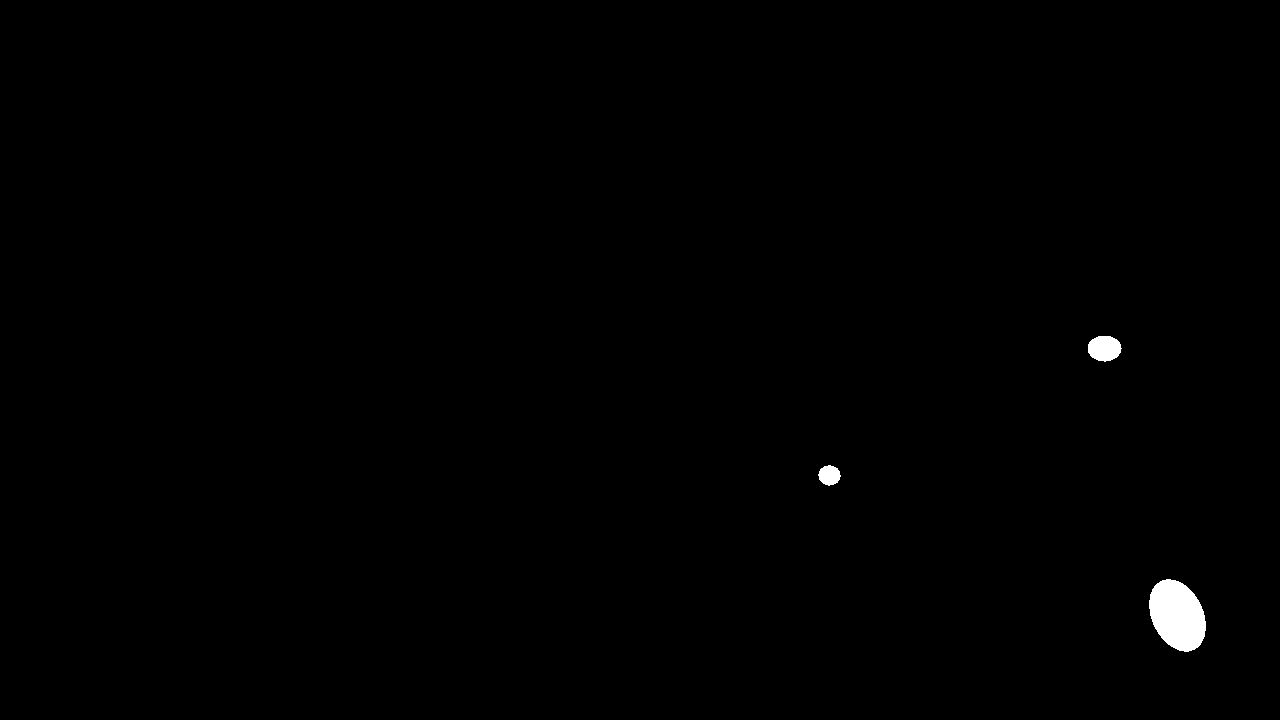


Figure 19 GT Image of frame-111

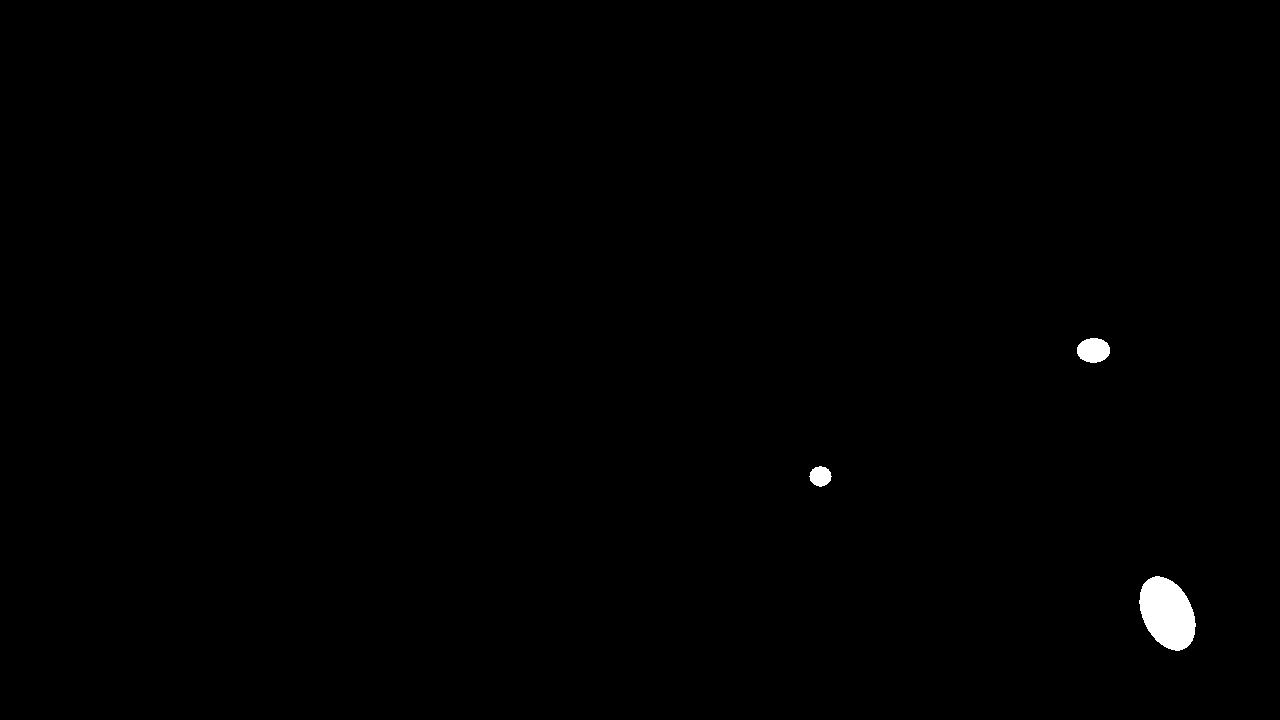


Figure 20 GT Image of frame-112

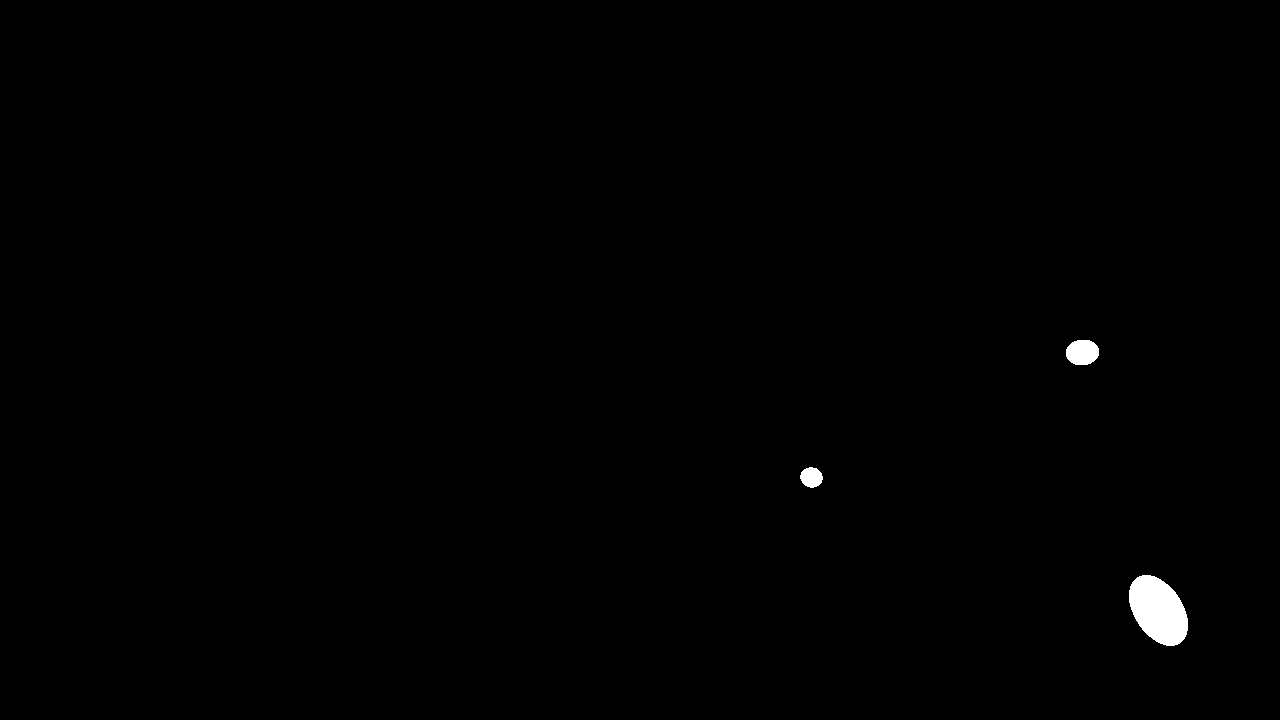


Figure 21 GT Image of frame-113

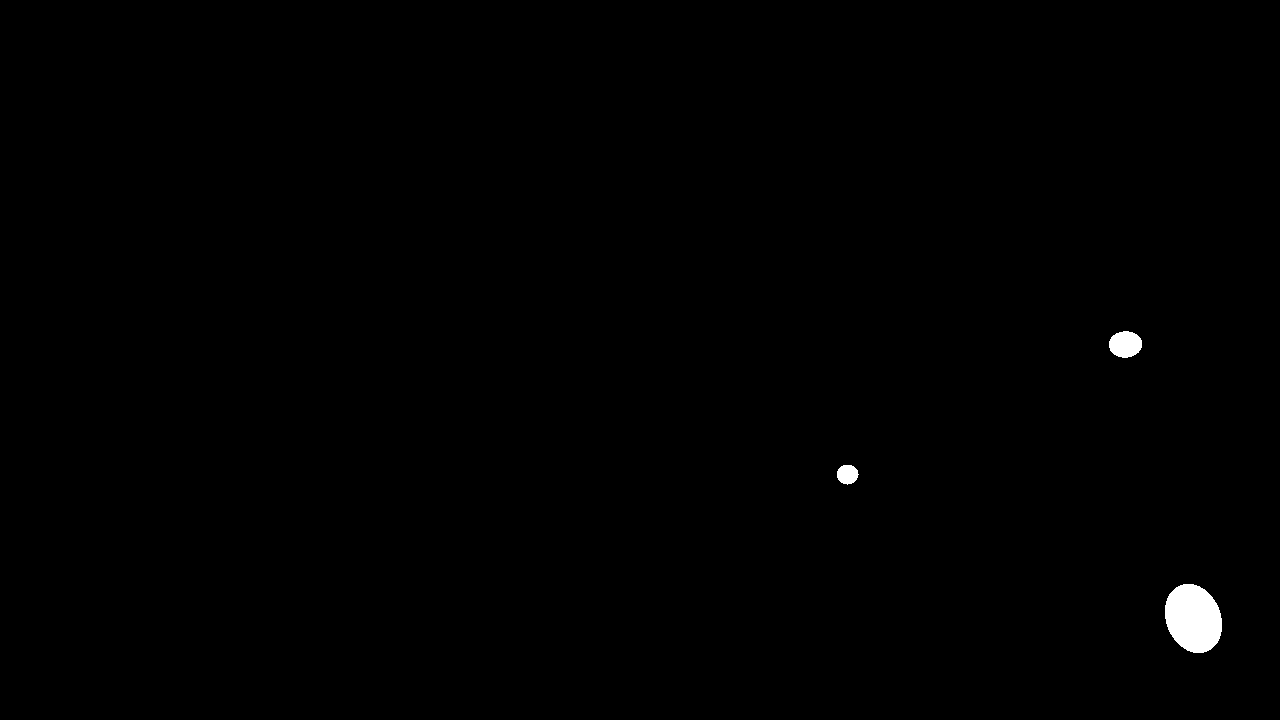


Figure 22 GT Image of frame-114

**Ground Truth Images of 5 worst Segmented Images:**

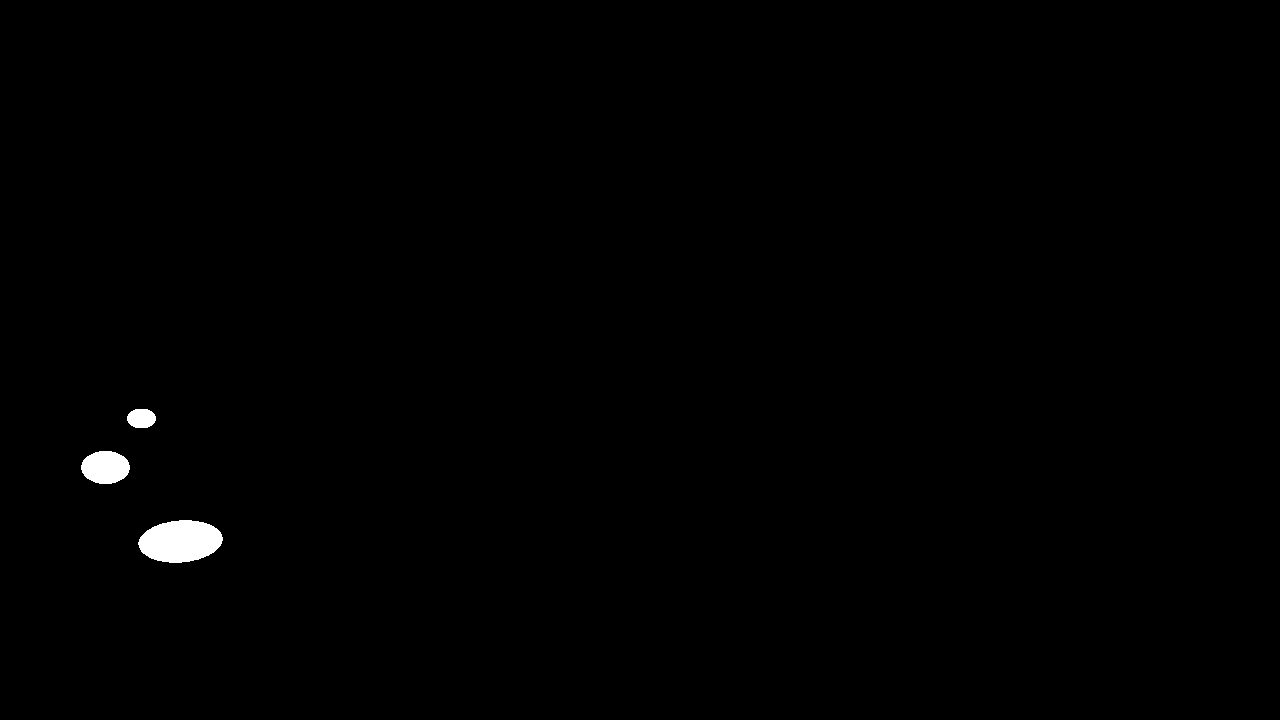


Figure 23 GT Image of frame-56

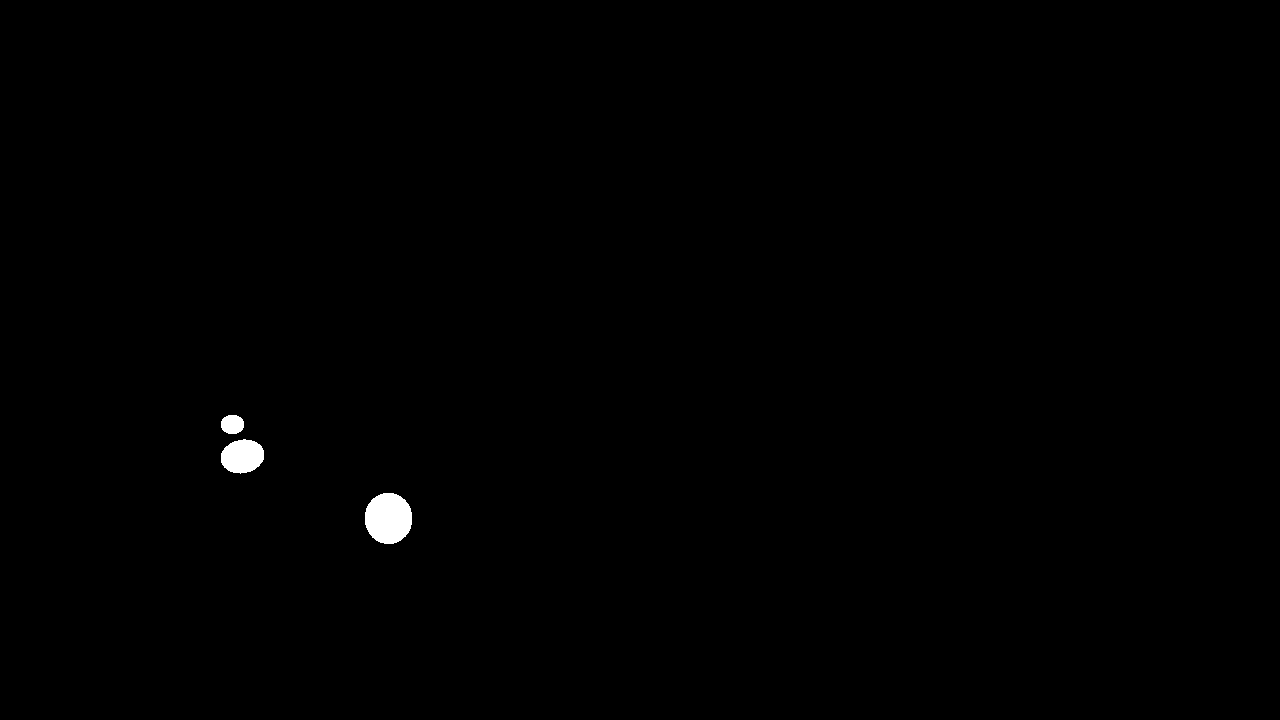


Figure 24 GT Image of frame-59

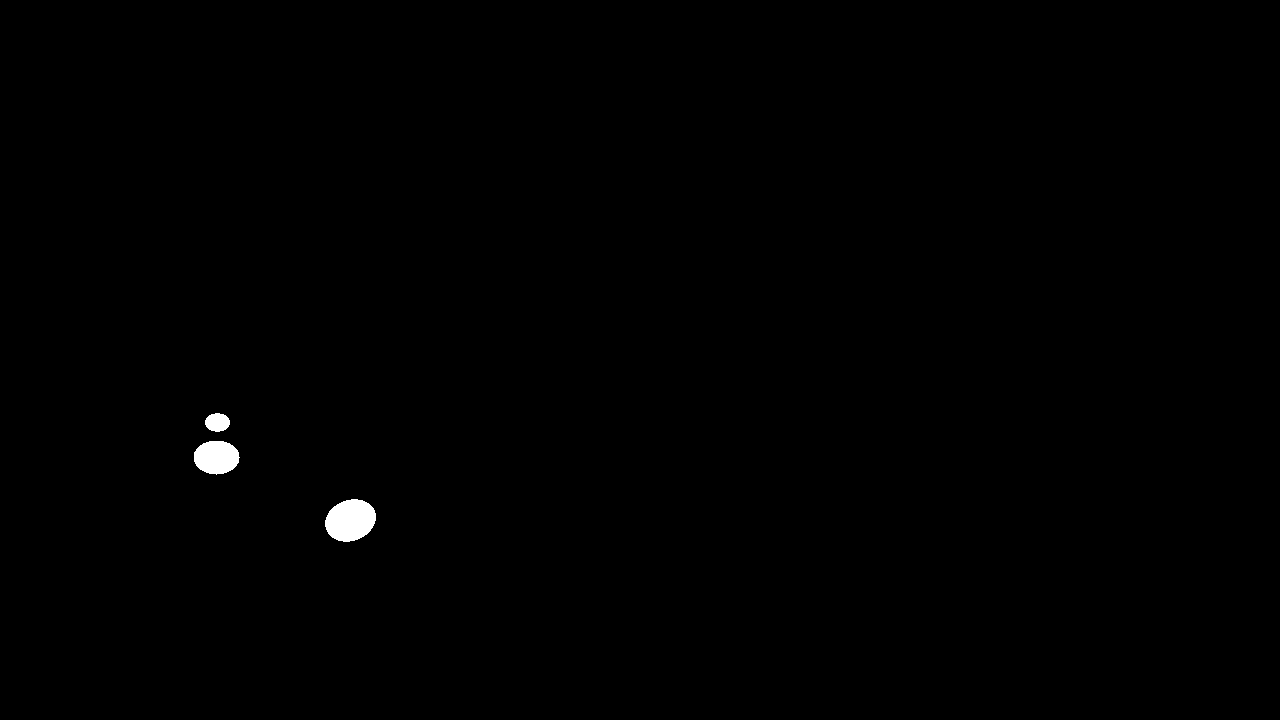


Figure 25 GT Image of frame-60

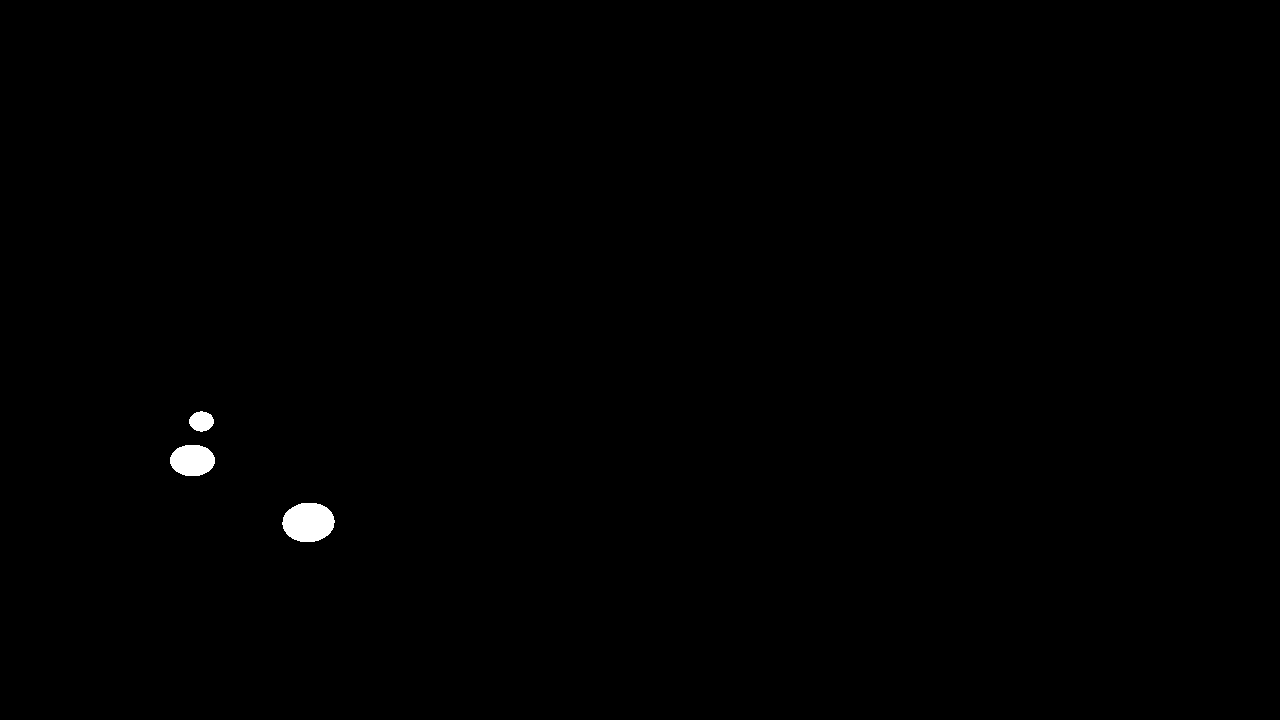


Figure 26 GT Image of frame-61

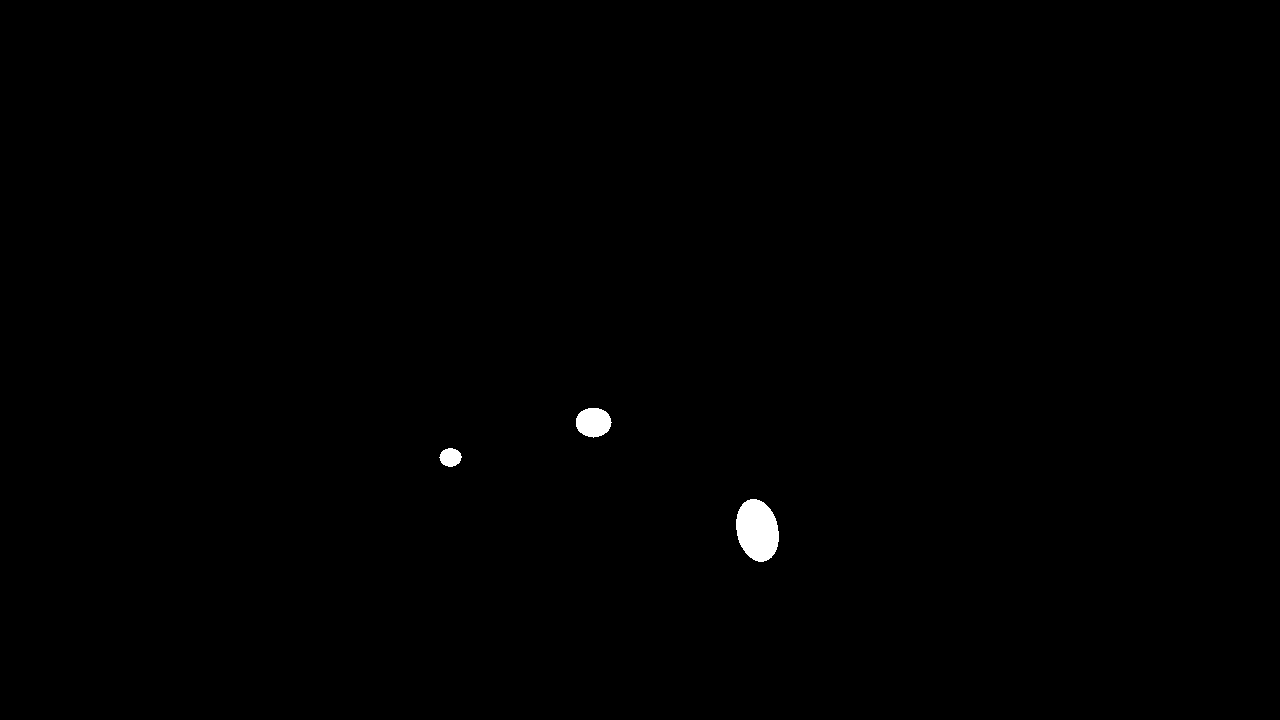


Figure 27 GT Image of frame-77

**Appendix B**

**Code of all the Task i.e. (Object Segmentation, Feature Extraction and Object Tracking)**

## Importing Libraries

import os

import cv2

import numpy as np

from google.colab.patches import cv2\_imshow

import matplotlib.pyplot as plt

from skimage.feature import graycomatrix, graycoprops

import skimage

import pandas as pd

Task 1: Image Segmentation and Detection

## A: Object segmentation

folder\_path = "/content/drive/MyDrive/ball\_frames/frames\_feature\_extraction\_image\_segmentation"

for image\_name in os.listdir(folder\_path):

    image\_path = os.path.join(folder\_path, image\_name)

    image\_name\_only, \_ = os.path.splitext(image\_name)

    image = cv2.imread(image\_path)

    blur = cv2.GaussianBlur(image, (7,7), 0)

    kernel = np.ones((5, 5), np.uint8)

    dilated = cv2.dilate(blur, kernel, iterations=1)

    # gray = cv2.cvtColor(image, cv2.COLOR\_BGR2HSV)

    edges = cv2.Canny(dilated, threshold1=0, threshold2=150)

    dilated\_edges = cv2.dilate(edges, kernel, iterations=1)

    contours, \_ = cv2.findContours(dilated\_edges, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

    # Get the height of the image

    height = image.shape[0]

    # Calculate the y-coordinate of the middle of the image

    middle\_y = (height // 2)-60

    # Create a black image with the same dimensions as the original image

    mask = np.zeros\_like(image)

    # Iterate through contours and draw only if centroid is below middle

    for contour in contours:

        # Calculate the centroid of the contour

        M = cv2.moments(contour)

        if M["m00"] != 0:

            cX = int(M["m10"] / M["m00"])

            cY = int(M["m01"] / M["m00"])

            # Check if centroid is below middle

            if cY > middle\_y:

                cv2.drawContours(mask, [contour], -1, (255, 255, 255), thickness=cv2.FILLED)

    # Show the masked image

    masked\_image = cv2.bitwise\_and(image, mask)

    # Convert the masked image to binary

    \_, binary\_image = cv2.threshold(masked\_image, 0, 255, cv2.THRESH\_BINARY)

    cv2.imwrite(f"/content/drive/MyDrive/ball\_frames/masked\_images/{image\_name\_only}\_masked.png", binary\_image)

print("Masked Images Saved to /content/drive/MyDrive/ball\_frames/masked\_images/")

## B: Segmentation evaluation

def dice\_similarity\_score(y\_true, y\_pred):

    intersection = np.sum((y\_true \* y\_pred))

    return (2. \* intersection) / (np.sum(y\_true==255) + np.sum(y\_pred==255))

# Ground-truth ball mask folder path

ground\_truth\_folder = "/content/drive/MyDrive/ball\_frames/gt"

# Segmented ball region folder path (output folder from Task 1)

segmented\_folder = "/content/drive/MyDrive/ball\_frames/masked\_images"

# Lists to store DS values and image numbers

ds\_values = []

image\_numbers = []

# Sort file names to ensure sequential processing

gt\_image\_names = sorted(os.listdir(ground\_truth\_folder))

segmented\_image\_names = sorted(os.listdir(segmented\_folder))

# Calculate DS for each ball image

for gt\_image\_name, segmented\_image\_name in zip(gt\_image\_names, segmented\_image\_names):

    gt\_image\_path = os.path.join(ground\_truth\_folder, gt\_image\_name)

    segmented\_image\_path = os.path.join(segmented\_folder, segmented\_image\_name)

    # Extract image number from ground truth image name

    image\_number = int(gt\_image\_name.split('-')[1].split('.')[0].split('\_')[0])

    # Read the ground truth and segmented images

    ground\_truth\_image = cv2.imread(gt\_image\_path, cv2.IMREAD\_GRAYSCALE)

    segmented\_image = cv2.imread(segmented\_image\_path, cv2.IMREAD\_GRAYSCALE)

    # Compute the Dice Similarity Score (DS)

    ds = dice\_similarity\_score(ground\_truth\_image, segmented\_image)

    print("DS Score between",gt\_image\_name,segmented\_image\_name,":",ds)

    # Append DS value and image number to lists

    ds\_values.append(ds)

    image\_numbers.append(image\_number)

# Plot bar graph

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

plt.bar(image\_numbers, ds\_values, color='skyblue')

plt.xlabel('Image Number')

plt.ylabel('Dice Similarity Score (DS)')

plt.title('Dice Similarity Score (DS) for Each Ball Image')

plt.xticks(image\_numbers,rotation='vertical')

plt.grid(axis='y')

plt.show()

# Calculate mean and standard deviation of DS

mean\_ds = np.mean(ds\_values)

std\_dev\_ds = np.std(ds\_values)

print("Mean DS:", mean\_ds)

print("Standard Deviation of DS:", std\_dev\_ds)

# Sort DS values and corresponding image numbers

sorted\_ds\_image\_pairs = sorted(zip(ds\_values, image\_numbers), reverse=True)

# Best 5 segmented ball images

print("\nBest 5 Segmented Ball Images:")

for ds, image\_number in sorted\_ds\_image\_pairs[:5]:

    print(f"Image Number: {image\_number}, DS: {ds}")

# Worst 5 segmented ball images

print("\nWorst 5 Segmented Ball Images:")

for ds, image\_number in sorted\_ds\_image\_pairs[-5:]:

    print(f"Image Number: {image\_number}, DS: {ds}")

## Task 2: Task 2: Feature Calculation

## A: Shape Features

def calculate\_shape\_features(contour):

    # Calculate contour of the object

    # Calculate area

    area = cv2.contourArea(contour)

    # Calculate perimeter

    perimeter = cv2.arcLength(contour, closed=True)

    # Calculate solidity

    solidity = area / perimeter\*\*2

    # Calculate non-compactness

    non\_compactness = perimeter\*\*2 / (4 \* np.pi \* area)

    # Calculate circularity

    circularity = (4 \* np.pi \* area) / perimeter\*\*2

    # Calculate eccentricity

    ellipse = cv2.fitEllipse(contour)

    (xc, yc), (d1, d2), angle = ellipse

    eccentricity = np.sqrt(1 - (d1\*\*2 / d2\*\*2))

    return solidity, non\_compactness, circularity, eccentricity

# Load original RGB images

# Directory paths

rgb\_images\_dir = '/content/drive/MyDrive/ball\_frames/frames\_feature\_extraction\_image\_segmentation'

gt\_masks\_dir = '/content/drive/MyDrive/ball\_frames/gt'

# Get full file paths for RGB images

rgb\_image\_paths = [os.path.join(rgb\_images\_dir, filename) for filename in sorted(os.listdir(rgb\_images\_dir))]

# Get full file paths for ground truth masks

gt\_mask\_paths = [os.path.join(gt\_masks\_dir, filename) for filename in sorted(os.listdir(gt\_masks\_dir))]

# Load original RGB images

rgb\_images = [cv2.imread(path) for path in rgb\_image\_paths]

# Load ground truth ball masks

gt\_masks = [cv2.imread(path, cv2.IMREAD\_GRAYSCALE) for path in gt\_mask\_paths]

# Iterate over each image and corresponding mask

for rgb\_image, gt\_mask in zip(rgb\_images, gt\_masks):

    # Extract ball patch from RGB image using the mask

    ball\_patch = cv2.bitwise\_and(rgb\_image, rgb\_image, mask=gt\_mask)

    # Convert mask to binary

    \_, binary\_mask = cv2.threshold(gt\_mask, 128, 255, cv2.THRESH\_BINARY)

    # Find contours in the binary mask and sort them based on area

    contours, \_ = cv2.findContours(binary\_mask, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

    contours = sorted(contours, key=cv2.contourArea)

    # Separate contours based on ball type (assuming there are three contours)

    pingpong\_ball\_contour = contours[0]

    football\_contour = contours[1]

    american\_football\_contour = contours[2]

    # Calculate shape features for each ball type

    pingpong\_solidity, pingpong\_non\_compactness, pingpong\_circularity, pingpong\_eccentricity = calculate\_shape\_features(pingpong\_ball\_contour)

    football\_solidity, football\_non\_compactness, football\_circularity, football\_eccentricity = calculate\_shape\_features(football\_contour)

    american\_football\_solidity, american\_football\_non\_compactness, american\_football\_circularity, american\_football\_eccentricity = calculate\_shape\_features(american\_football\_contour)

    # Plot distribution of shape features for each ball type

    features\_lists = [

        (pingpong\_solidity, football\_solidity, american\_football\_solidity),

        (pingpong\_non\_compactness, football\_non\_compactness, american\_football\_non\_compactness),

        (pingpong\_circularity, football\_circularity, american\_football\_circularity),

        (pingpong\_eccentricity, football\_eccentricity, american\_football\_eccentricity)

    ]

    feature\_names = ['Solidity', 'Non-compactness', 'Circularity', 'Eccentricity']

    ball\_types = ['Pingpong Ball', 'Football', 'American Football']

    colors = ['skyblue', 'salmon', 'lightgreen']

    plt.figure(figsize=(12, 3))

    for i, features in enumerate(features\_lists):

        plt.subplot(1, 4, i+1)

        for j, feature in enumerate(features):

            plt.hist(feature , bins=20, color=colors[j], edgecolor='black', alpha=0.5, label=ball\_types[j])

        plt.xlabel(feature\_names[i])

        plt.ylabel('Frequency')

        plt.title(f'Distribution of {feature\_names[i]}')

        plt.legend()

    plt.tight\_layout()

    plt.show()

## B: Texture Features

def calculate\_texture\_features(patch):

    # Calculate GLCM for each color channel and each orientation

    glcms = []

    for channel in range(patch.shape[2]):

        glcm\_0 = graycomatrix(patch[:, :, channel], [1], [0], symmetric=True, normed=True)

        glcm\_45 = graycomatrix(patch[:, :, channel], [1], [np.pi / 4], symmetric=True, normed=True)

        glcm\_90 = graycomatrix(patch[:, :, channel], [1], [np.pi / 2], symmetric=True, normed=True)

        glcm\_135 = graycomatrix(patch[:, :, channel], [1], [3 \* np.pi / 4], symmetric=True, normed=True)

        glcms.extend([glcm\_0, glcm\_45, glcm\_90, glcm\_135])

    # Calculate Haralick features for each orientation

    haralick\_features = []

    for glcm in glcms:

        haralick\_features.append([

            graycoprops(glcm, 'ASM')[0, 0],

            graycoprops(glcm, 'contrast')[0, 0],

            graycoprops(glcm, 'correlation')[0, 0]

        ])

    # Calculate feature average and range across orientations

    haralick\_features = np.array(haralick\_features)

    feature\_avg = np.mean(haralick\_features, axis=0)

    feature\_range = np.ptp(haralick\_features, axis=0)

    return haralick\_features,feature\_avg, feature\_range

# Iterate over each image and corresponding mask

for rgb\_image, gt\_mask in zip(rgb\_images, gt\_masks):

    contours, \_ = cv2.findContours(gt\_mask, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

    # Sort contours based on area

    contours = sorted(contours, key=cv2.contourArea)

    # Extract ball patches for each ball type using the contours

    pingpong\_ball\_patch = cv2.bitwise\_and(rgb\_image, rgb\_image, mask=np.zeros\_like(gt\_mask))

    football\_patch = cv2.bitwise\_and(rgb\_image, rgb\_image, mask=np.zeros\_like(gt\_mask))

    american\_football\_patch = cv2.bitwise\_and(rgb\_image, rgb\_image, mask=np.zeros\_like(gt\_mask))

    for i, contour in enumerate(contours):

        if i == 0:  # Smallest contour (pingpong ball)

            cv2.drawContours(pingpong\_ball\_patch, [contour], -1, (255, 255, 255), thickness=cv2.FILLED)

        elif i == 1:  # Second smallest contour (football)

            cv2.drawContours(football\_patch, [contour], -1, (255, 255, 255), thickness=cv2.FILLED)

        elif i == 2:  # Largest contour (american football)

            cv2.drawContours(american\_football\_patch, [contour], -1, (255, 255, 255), thickness=cv2.FILLED)

    # Calculate texture features for each ball type

    pingpong\_features, pingpong\_avg, pingpong\_range = calculate\_texture\_features(pingpong\_ball\_patch)

    football\_features, football\_avg, football\_range = calculate\_texture\_features(football\_patch)

    american\_football\_features, american\_football\_avg, american\_football\_range = calculate\_texture\_features(american\_football\_patch)

    # Select one feature from each color channel

    # For example, let's select Angular Second Moment from each color channel

    selected\_features = [

        (pingpong\_features[:, 0] + football\_features[:, 0] + american\_football\_features[:, 0]) / 3,  # ASM average

        (pingpong\_features[:, 1] + football\_features[:, 1] + american\_football\_features[:, 1]) / 3,  # Contrast average

        (pingpong\_features[:, 2] + football\_features[:, 2] + american\_football\_features[:, 2]) / 3,  # Correlation average

    ]

    # Plot distribution per ball type for selected features

    feature\_names = ['Angular Second Moment', 'Contrast', 'Correlation']

    ball\_types = ['Pingpong Ball', 'Football', 'American Football']

    plt.figure(figsize=(12, 3))

    for i, feature in enumerate(selected\_features):

        plt.subplot(1, 3, i+1)

        for j in range(3):  # Loop over ball types

            plt.hist(feature[j], bins=20, alpha=0.5, label=ball\_types[j])

        plt.xlabel(feature\_names[i])

        plt.ylabel('Frequency')

        plt.title(f'Distribution of {feature\_names[i]}')

        plt.legend()

    plt.tight\_layout()

    plt.show()

## Task 3: Object Tracking

# Load data

x = pd.read\_csv('/content/drive/MyDrive/ball\_frames/x.csv', header=None)

y = pd.read\_csv('/content/drive/MyDrive/ball\_frames/y.csv', header=None)

na = pd.read\_csv('/content/drive/MyDrive/ball\_frames/na.csv', header=None)

nb = pd.read\_csv('/content/drive/MyDrive/ball\_frames/nb.csv', header=None)

# Extract actual data values

x\_data = x.iloc[0, :]

y\_data = y.iloc[0, :]

na\_data = na.iloc[0, :]

nb\_data = nb.iloc[0, :]

# Constants and Matrices Initialization

dt = 0.5

F = np.array([[1, dt, 0, 0], [0, 1, 0, 0], [0, 0, 1, dt], [0, 0, 0, 1]])

H = np.array([[1, 0, 0, 0], [0, 0, 1, 0]])

Q = np.array([[0.16, 0, 0, 0], [0, 0.36, 0, 0], [0, 0, 0.16, 0], [0, 0, 0, 0.36]])

R = np.array([[0.25, 0], [0, 0.25]])

# Adjusted starting estimate of the state

x\_hat = np.array([na\_data[0],(na\_data[1] - na\_data[0]) / dt, nb\_data[0], (nb\_data[1] - nb\_data[0]) / dt])

# Adjusted initial covariance estimate, increase uncertainty in velocity

P = np.array([[10 , 0, 0, 0], [0, 10, 0, 0], [0, 0, 10, 0], [0, 0, 0, 10]])

# Simulation of the Kalman Filter

n = len(na\_data)  # Number of measurements

estimates = []

for i in range(n):

    z = np.array([na\_data[i], nb\_data[i]])

    x\_hat = F.dot(x\_hat)

    P = F.dot(P).dot(F.T) + Q

    S = H.dot(P).dot(H.T) + R

    K = P.dot(H.T).dot(np.linalg.inv(S))

    y = z - H.dot(x\_hat)

    x\_hat = x\_hat + K.dot(y)

    P = (np.eye(4) - K.dot(H)).dot(P)

    estimates.append([x\_hat[0], x\_hat[2]])

    print(f"Iteration {i}:\nActual", "[", na\_data[i], nb\_data[i], "]")

    print(f"Estimate {[x\_hat[0], x\_hat[2]]}")

estimates = np.array(estimates)

# Plotting

plt.figure(figsize=(10, 8))

plt.plot(x\_data, y\_data, label='True Position')

plt.scatter(na\_data, nb\_data, color='red', label='Noisy Measurements', alpha=0.5)

plt.plot(estimates[:, 0], estimates[:, 1], label='Kalman Filter Estimate')

plt.title('Kalman Filter Tracking')

plt.xlabel('X Position')

plt.ylabel('Y Position')

plt.legend()

plt.show()

# Calculate RMSE

true\_positions = np.column\_stack((x\_data, y\_data))

rmse = np.sqrt(np.mean((estimates - true\_positions)\*\*2, axis=0))

print(f"RMSE: {rmse}")