Computer Vision – Assignment 1

Omar Ali - 28587497

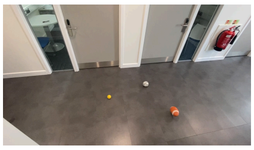
2024-05-07

1: Image Segmentation and Detection

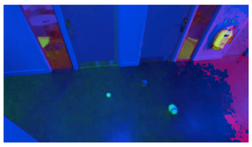
In this task, 3 balls from some images needed to be extracted from a set of 62 images and segmented into a mask. The balls were of three types: American Football, Football, and Tennis Ball. The images represent a sequence or frames where the balls are in motion. The task was to segment the balls from the background and calculate the Dice Similarity Score (DS) of the segmented balls against the ground truth mask of the balls.

1.1 Ball Segmentation

In this section, I will demonstrate how the balls were segmented from the background using a series of image-processing techniques. The steps are as follows:



Original
Step 1
Grab the original image from path

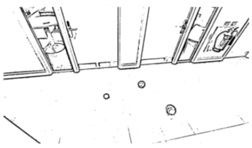


HSV
Step 2
Converted the RGB image into HSV colour space

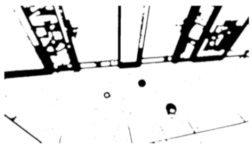
Intensity
Step 3
Extracted the Intensity Channel

GBlur
Step 4
Apply Gaussian Blur with k=[3x3] with 2 iterations

Median Blur
Step 5
Apply Median Blur with k=[3x3] with 2 iterations

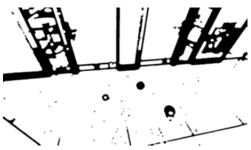


Adaptive Gaussian Threshold
Step 6
With Blocksize = 19 and c = 5

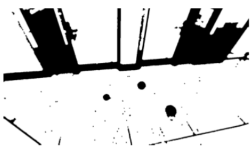


Opening
Step 7
to disconnecting white pixels k = (4,4) with 5 iterations

Dilate
Step 8
expanding white pixels to remove noisy black holes holes k = (3,3) with 4 iterations



Erode
Step 9
re-shrinking the white pixels to let the balls begin to connect k = (3,3) with 2 iterations



Fill
Step 10
Fill the enclosed black areas

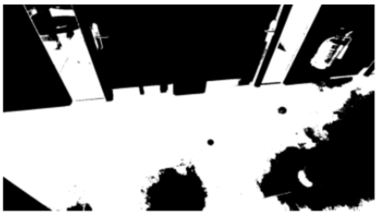
Dilate and Erode
Step 11
series of dilations and erosions to remove the noisy black pixels whilst maintaining *

Contour
Step 12
find the contours of the (bitwise_not) image

At this stage, step 12, it can be seen that the balls have been segmented from the background. However, this is only the case in the particular sample, frame 85. In the images below, Frame 100 is re-displayed to show that the ball segmentation still had artifacts that needed to be removed. Two main culprits were the shadows of the American Football in the latter frames, as well as some sections of the background that were not yet caught by the segmentation process so far.

In order to remove the shadow, it was found that the intensity channel of the HSV image was quite effective and computationally cheap to apply.

Contour
Step 12
Grab the original image from path



Thresholing Intensity
Step 13.1
A threshold of 0.4 was applied to the intensity image from step 3 to remove shadows

Combine
Step 13.2
The contour and the thresholded intensity image were combined

The inverse of the thresholded intentity, in step 13.1, was combined with the contour using a 'bitwise and' operation.

```
combo = cv2.bitwise_not(cv2.bitwise_or(thresh_intensity_image, contour_image))
```

Convex Hull
Step 14

Circularity
Step 15

Finally, a convex hull was applied to the contours to remove sharp edges and improve the circularity of the contours. The circularity of the contours was calculated and only contours with a circularity greater than 0.72 were kept.

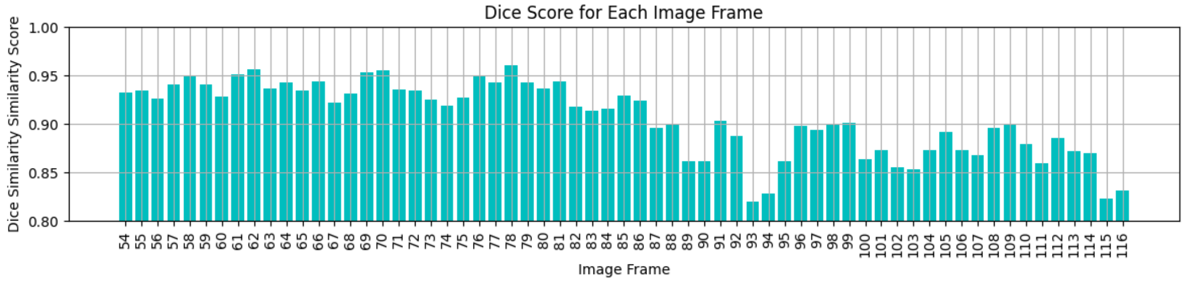
1.2 Dice Similarity Score

The Dice Similarity Score (DS) of every segmented ball image was compared against the ground truth mask of the ball image. The Dice Similarity Score (DS) is defined as:

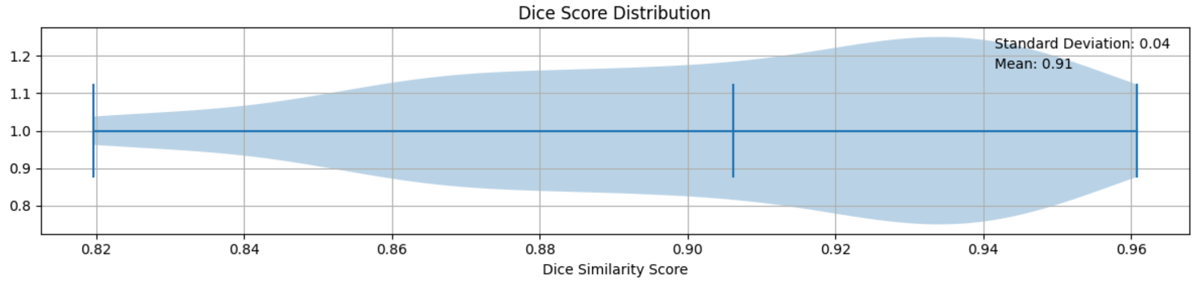
$$\mathrm{DS} = \frac{2 * |M \cap S|}{|M| + |S|}$$

where M is the ground truth mask and S is the segmented mask. The DS score ranges from 0 to 1, where 1 indicates a perfect match between the two masks.

A bar chart of the DS has been plotted for every frame. The lowest recorded score is 0.82 for frame 93 and the highest is 0.96 for frame 78.



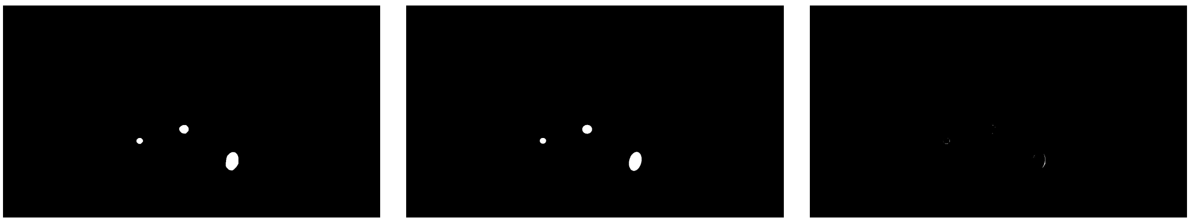
Additionally, a violin plot, was used to display the distribution of the DS score across all the frames, it can be seen that there is quite a skew of the data towards the higher range of the DS score, with the greatest proportion of the data sitting around the 0.93-0.94 range.



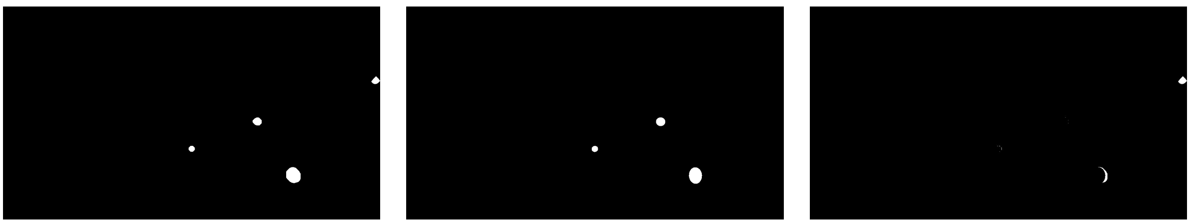
Below are the best and worst frames in terms of the Dice Similarity Score achieved by the segmentation process.

Best Frame 78 - DS 0.961

Segmented Image Ground Truth Difference



Worst Frame 78 - DS 0.961



Segmented Image 93 Ground Truth 93 DS 0.820

1.3 Discussion

The implementation was found to be quite good, where the DS scores had a maximum of 0.96, minimum of 0.82, mean of 0.91 and standard deviation of 0.04. However, the implementation is quite clunky and very tuned for this task.

Overall, the number of steps in order to reach the solution is quite large, at 15 steps. The additional processing was ultimately due to a whack-a-mole situation, where refinements in one area of the task cause another area to worsen. This makes this solution temperamental and not very robust to changes in the input data, however, for very refined results, this is likely to be a similar case for most image segmentation methodologies that do not more complex models such as deep learning.

In the initial processing, it was found that the Intensity value provided a very good initial starting point, as opposed to using grayscale, as the intensity of the balls has a very distinct value compared to the background. The Gaussian Blur and Median Blur were very effective at removing the nose from this image, whilst maintaining the edges of the balls. The Adaptive Gaussian Threshold (AGT) was effective in detecting the edges in the image, compared to a standard threshold, because it takes into consideration a normalised local area for its thresholding calculation.

The range of morphological filters was essential in getting a cleaner mask of the balls, with just the right amount of erosion and dilation to remove the noise from the background. Different ranges of kernel sizes were used for the erosion and dilation, however all used some scale of the `MORPH_ELLIPSE` structuring element. It is important to note that in the first applications of the morphological filters, the region being eroded or dilated was the background, not the balls. This was because the image had not yet been inverted and so the balls were considered the background.

Once the morphological filtering was complete, the balls were segmented using the `opencv.findContours` function. As mentioned, this was found to not be refined enough, so the intensity channel was used to remove the shadows from the American Football, a convex hull was applied to the contours to connect sharp edges, and the circularity of the contours where only contours with a circularity greater than 0.72 were kept.

There are certainly some improvements that could be made, and would have possibly decided to take a different approach had this task been reattempted. A possible alternative first step could be to make use of the colour channels of the image to segment the balls. This could be done by applying a threshold to the colour channels and then combining the results. This would have been a more robust approach where the main challenge would be the distinction of the white football from the walls of the room. Additionally, considering the constrained environment, it could have been possible to create a mask of the problematic parts of the room and have these be removed from the image before segmentation.

2: Feature Calculation

2.1 Shape Features

In this section, I will demonstrate how a range of shape features (Solidity, Non-compactness, Circularity, Eccentricity) can be calculated from the contours of the segmented balls.

In order to get the contours of the segmented balls, I made use of the openCV 'findContours' function which returns a list of contours and a hierarchy for all the contours in the image.

```
img = cv2.imread(image)
img = cv2.bitwise_and(cv2.cvtColor(msk, cv2.COLOR_GRAY2BGR), img)
contours, _ = cv2.findContours(msk, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
```

Each contour is split into the corresponding balls in order to combine the contours of the same ball into an array, this is done by exploiting the areas of each ball. The details of the code can be explored as needed, however the core logic is as follows:

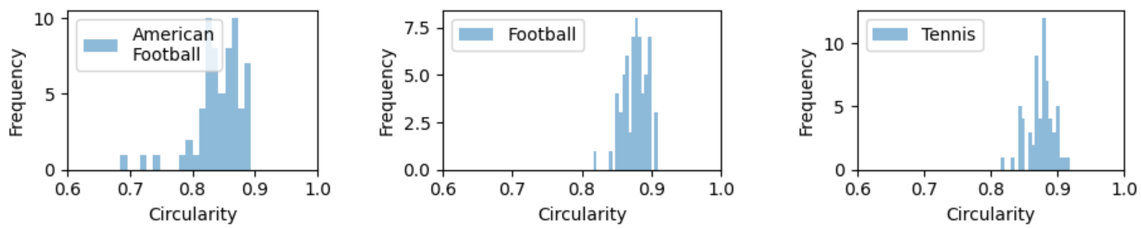
```python
area = cv2.contourArea(contour)
if area > 1300:  # football
    append_ball = BALL_LARGE
elif area > 500:  # soccer_ball
    append_ball = BALL_MEDIUM
else:  # tennis ball
    append_ball = BALL_SMALL
```

Using this, the features can be evaluated for every ball.

2.1.a Circularity

Circularity is defined as the ratio of the area of the circle with the same perimeter as the object to the area of the object. For a given contour C, the circularity is calculated by:

```python
area = cv2.contourArea(contour)
perimeter = cv2.arcLength(contour, closed=True)
circularity = (4 * math.pi * area) / (perimeter**2)
```

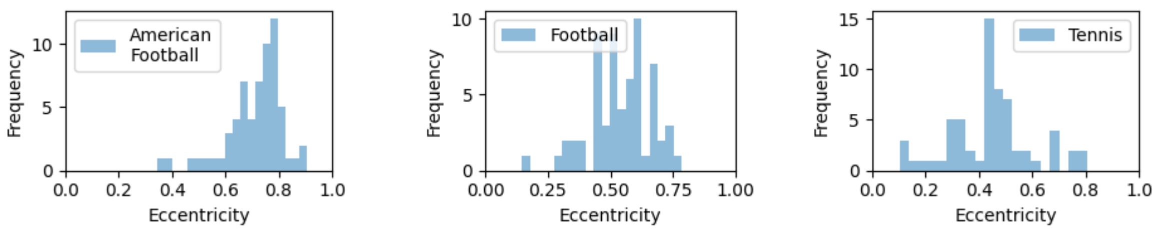


Both the Football and the Tennis ball have a higher circularity than the American Football. This is expected as the American Football has a more elongated shape compared to the other two balls. Visually, the football has a smaller variance in circularity compared to the tennis ball. This is likely due to the relative size of the football compared to the tennis ball, where the football is larger and has a more consistent shape from the perspective of an image and will not suffer from distortion and be impacted by smaller pixel ranges.

2.1.b Eccentricity

Eccentricity is the ratio of the distance between the foci of the ellipse to the major axis length. For a given contour C, the eccentricity is calculated by:

```python
ellipse = cv2.fitEllipse(contour)
a = max(ellipse[1])
b = min(ellipse[1])
eccentricity = (1 - (b**2) / (a**2)) ** 0.5
```

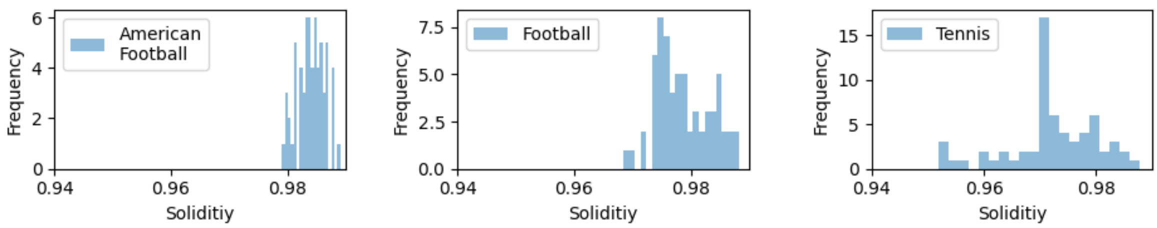


The American Football has the highest eccentricity of all the balls, which is expected as it has a more elongated shape compared to the other two balls. The football and the tennis ball both have very distributed eccentricity values.

2.1.c Solidity

Solidity is the ratio of the area of the object to the area of the convex hull of the object. For a given contour C, the solidity is calculated by:

```
area = cv2.contourArea(contour)
convex_hull = cv2.convexHull(contour)
convex_area = cv2.contourArea(convex_hull)
solidity = area / convex_area
```

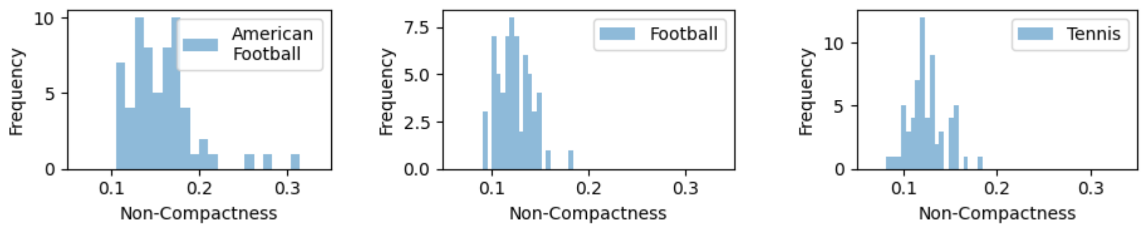


The solidity of the American Football is much higher and more consistent than the other two balls. This is likely due to the ball being larger in size and so a convex hull around the ball is likely to be more similar to the ball itself. This follows through as we see the football having a higher solidity than the tennis ball, and the tennis ball's solidity is much more distributed than the others.

2.1.d Non-compactness

Non-compactness is the ratio of the area of the object to the area of the circle with the same perimeter as the object. For a given contour C, the non-compactness is calculated by:

```
area = cv2.contourArea(contour)
perimeter = cv2.arcLength(contour, closed=True)
non_compactness = 1 - (4 * math.pi * area) / (perimeter**2)
```



The tennis ball has the tightest distribution of non-compactness values, with the American Football having the highest non-compactness values. This is because the American Football has a more elongated shape that changes its dimensions as the perspective shifts.

2.2 Texture Features

In this section, texture features are calculated from the segmented balls. The texture features are evaluated by calculating the normalised Grey-Level Co-occurrence Matrix (GLCM) in four orientations (0°, 45°, 90°, 135°) for every individual ball. The GLCM is a matrix that describes how often different combinations of pixel intensity values occur in an image. The GLCM is calculated for each of the colour channels (red, green, blue) and for each of the four orientations then averaged across the orientations to determine the texture features of the ball. These features include Angular Second Moment, Contrast, Correlation, and Entropy.

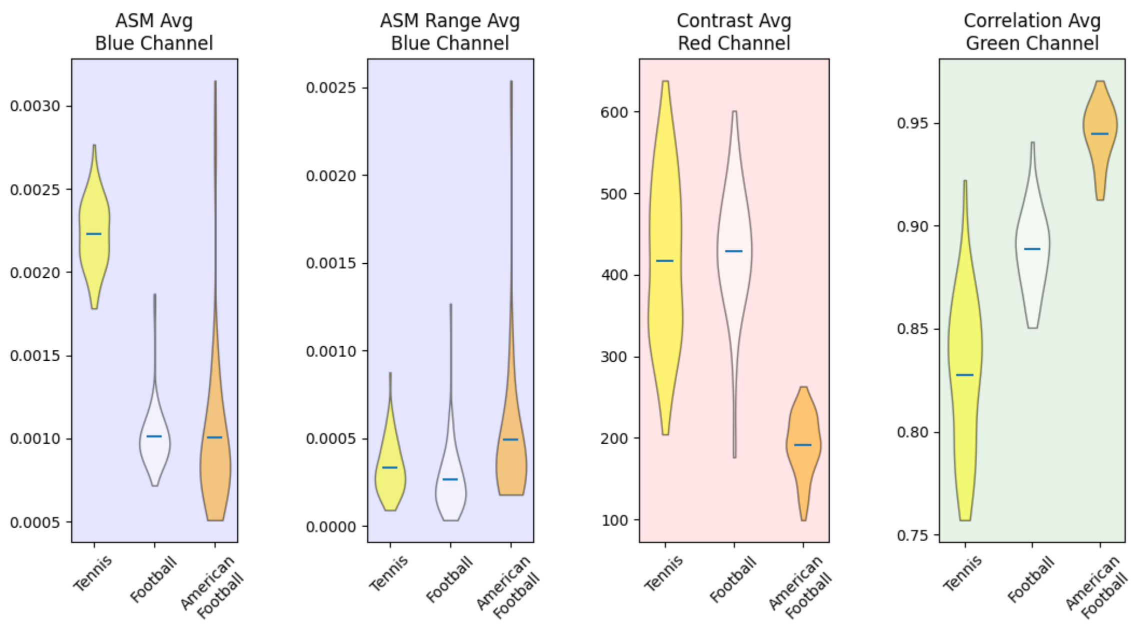
For each feature, one colour channel is selected to demonstrate the feature calculation. The blue channel was selected for the Angular Second Moment, the red channel for Contrast, and the green channel for Correlation.

To get the GLCM, the balls were segmented in a similar way to what was described previously in the shape features section. However, this time, the mask generated was overlayed onto the original image to get the pixel values of the balls.

The GLCM was calculated using the 'greycomatrix' function from the skimage library. The first row and column were both stripped away from the GLCM to remove the background noise. The GLCM was then normalised using 'greycoprops' and the texture features were evaluated for all four orientations. These were then averaged to get the final average texture value for the ball.

2.2.a Applied Texture Features

Below are a set of violin plot of the texture featuures for the three balls. The Angular Second Moment was calculated for the blue channel, the Contrast for the red channel, and the Correlation for the green channel, where the yellow plot represents the Tennis Ball, the white plot represents the Football, and the orange plot represents the American Football.



The **ASM**, is a measure of textural uniformity within an image. The ASM will be high when the image has constant or repetitive patterns, meaning the pixel intensities are similar or identical throughout the image. The yellow tennis ball has a high ASM mean in the blue channel, which suggests that the pixel intensities in the blue channel for the yellow tennis ball are very similar or identical throughout the image. This could be due to the yellow color having a low blue component in the RGB color model. The orange American football has a low ASM mean in the blue channel, which suggests that there is a lot of variation or randomness in pixel intensities in the blue channel for the orange American football. This could be due to the orange color having a low blue component in the RGB color model, or due to variations in lighting conditions or reflections on the ball. The large standard deviation indicates that the pixel intensities in the blue channel for the orange American football vary widely. This could be due to factors such as lighting conditions, shadows, or variations in the ball's color.

The **ASM range** in an image indicates the spread of textural uniformity across the image. A high range would indicate that there are areas of the image with very high textural uniformity. The yellow tennis ball has a low ASM range mean in the blue channel. This means that the pixel intensities in the blue channel for the yellow tennis ball are very similar or identical throughout the image.

The **contrast** of an image, specifically in a color channel, refers to the difference in color that makes an object distinguishable. In the red channel of an image, objects with a high amount of red will have a high intensity. The yellow tennis ball and the white football have the highest contrast means in the red channel because yellow, and white have a combination of red and green in the RGB color model.

The **correlation** of an image, specifically in a color channel, refers to the degree to which neighboring pixel values are related. In the green channel of an image, objects with a high amount of green will have a high intensity. The orange American football has a high correlation mean in the green channel, which suggests that it has a strong relationship with neighboring pixel values in the green channel. This could be due to the orange color having less green component compared to yellow or due to different lighting conditions. The yellow tennis ball has a low correlation mean in the green channel because yellow is a combination of red and green in the RGB color model. This means that the yellow tennis ball will have a high intensity in the green channel, but the correlation is low.

In terms of using the features to classify the balls, the Shape features can be very useful, especially in classifying the American Football as it has the most distinctive shape of the three balls. In particular, the solidity of the American Football has a very high value, and low distribution, compared to the others, making it a particularly distinguishing feature. The texture features can also be useful in identifying the balls but this is colour dependent. The tennis ball tends to have the lowest correlation and the American Football the highest.

3: Object Tracking

3.1 Kalman Filter

A Kalman filter is an algorithm that uses a series of measurements observed over time, containing statistical noise and other inaccuracies, and produces estimates of unknown variables that tend to be more accurate than those based on a single measurement alone.

It can be used in the context of object tracking to estimate the position of an object based on noisy measurements of its position. The Kalman filter uses a motion model to predict the next state of the object and an observation model to update the state based on the measurements.

The Kalman filter consists of two main steps: the prediction step and the update step.

These require a few sets of parameters to be set up, such as the state vector x, the state covariance matrix P, the motion model F, the observation model H, the process noise covariance matrix Q, and the measurement noise covariance matrix R.

```
x = np.matrix([xO]).T
Q = kq * np.matrix([[nx, 0, 0, 0], [0, nvx, 0, 0], [0, 0, ny, 0], [0, 0, 0, nvy]])
P = Q
F = np.matrix([[1, dt, 0, 0], [0, 1, 0, 0], [0, 0, 1, dt], [0, 0, 0, 1]])
H = np.matrix([[1, 0, 0, 0], [0, 0, 1, 0]])
R = kr * np.matrix([[nu, 0], [0, nv]])
N = len(z[0])
s = np.zeros((4, N))
```

Intial Parameters:

```
nx=0.16, ny=0.36, nvx=0.16, nvy=0.36, nu=0.25, nv=0.25
xO=[0,0,0,0], z=[], kr=1, kq = 1, dt=0.05
```

In the prediction step, the filter uses the motion model to predict the next state of the object based on the previous state.

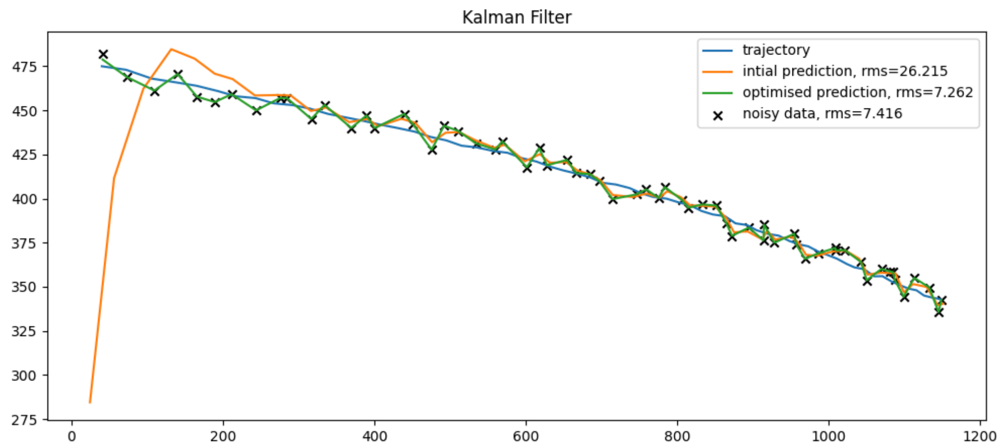
```
def kalman_predict(x, P, F, Q):
    xp = F * x
    Pp = F * P * F.T + Q
    return xp, P

def kalman_update(x, P, H, R, z):
    S = H * P * H.T + R
    K = P * H.T * np.linalg.inv(S)
    zp = H * x
    xe = x + K * (z - zp)
    Pe = P - K * H * P
    return xe, Pe
```

In the update step, the filter uses the observation model to update the state based on the measurements.

```python
for i in range(N):
        xp, Pp = kalman_predict(x, P, F, Q)
        x, P = kalman_update(xp, Pp, H, R, z[:, i])
        val = np.array(x[:2, :2]).flatten()
        s[:, i] = val
    px = s[0, :]
    py = s[1, :]
```

The plotted graph of the initial noisy coordinates [na,nb] and the estimated coordinates [x*,y*] can be seen below.



In order to evaluate the performance of the Kalman filter, the root mean squared error (RMSE) can be evaluated to show how close the estimated trajectory is to the ground truth trajectory.

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2}$$

```python
rms = np.sqrt(1/len(px)  * (np.sum((x - px)**2 + (y - py)**2)))
```

The initial Kalman filter implementation used a starting point of (0,0) for the x and y coordinates and had a mean of 9.68 and an RMS of 26.21. This performance was suboptimal compared to the noise, which had a mean of 6.96 and an RMS of 7.42.

The starting point was changed to the first point in the trajectory. The measurement noise parameter `nv` was set to be three magnitudes lower than `nu`, reflecting the smaller changes in the y positional values compared to the x positional values. The process noise parameters were also adjusted, with `nx` increased to 1.6 and `ny` to increase the noise on the x positional data which allows the filter to be more flexible in its predictions allowing for smoother transitions between the timesteps.

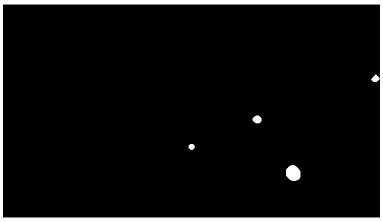
The final optimised parameters were set to:

```python
kq = 1.75E-2, kr = 1.5E-3, nx = 1.6, ny = 0.32, nvx = 0.16*1.75E-2
nvy = 0.36*1.75E-2, nu = 0.25, nv = 0.25E-3
```

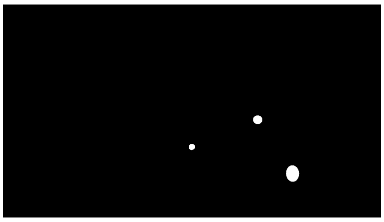
These adjustments resulted in a significant improvement in the filter's performance, with the mean reduced to 6.81 and the RMS reduced to 7.26, bringing the prediction closer to the ground truth.

4: Appendix.

4.1 Worst DS Frames



Segmentation-93



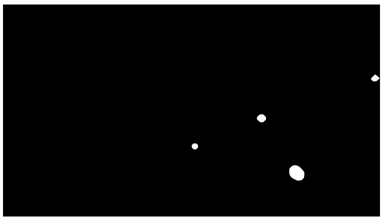
GT-93

Difference 0.820

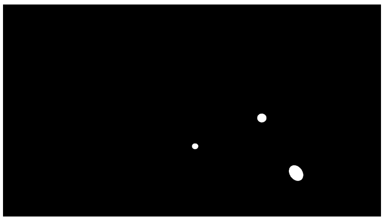
Segmentation-115

GT-115

Difference 0.823



Segmentation-94



GT-94

Difference 0.829

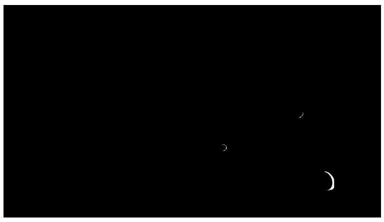
Segmentation-116

GT-116

Difference 0.831

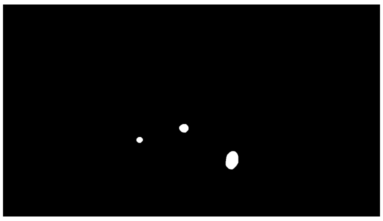
Segmentation-103

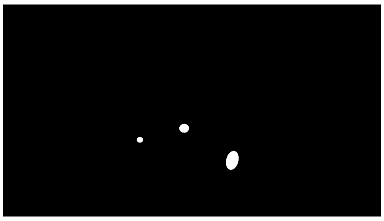
GT-103



Difference 0.853

4.2 Best DS Frames


Segmentation-78

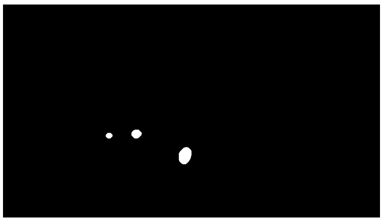

GT-78

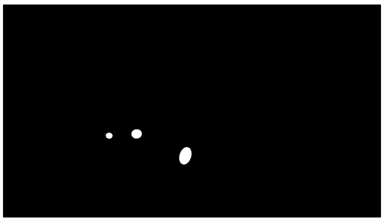

Difference 0.961


Segmentation-62


GT-62


Difference 0.957


Segmentation-70

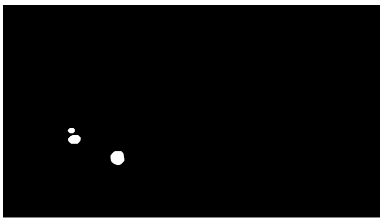

GT-70

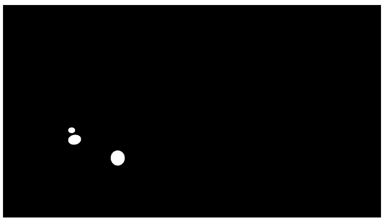

Difference 0.956

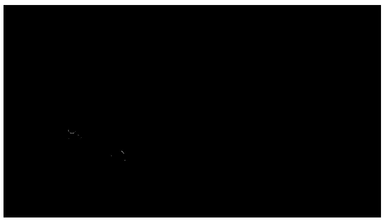

Segmentation-69


GT-69

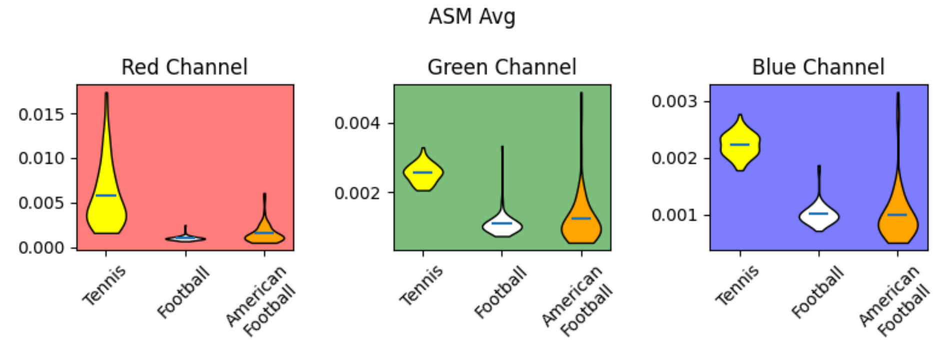

Difference 0.954

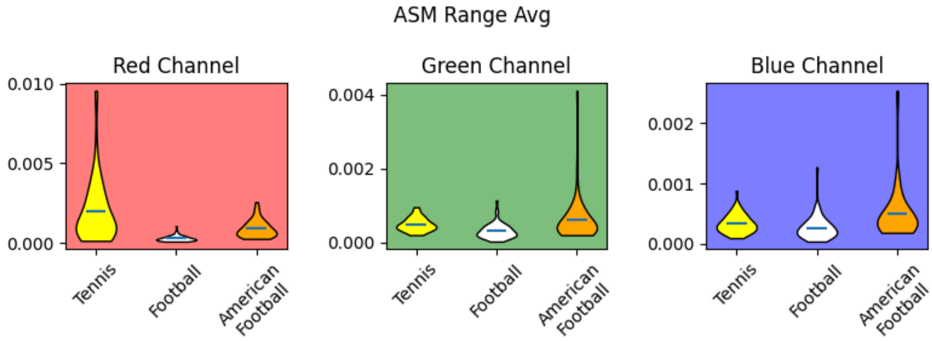

Segmentation-61


GT-61

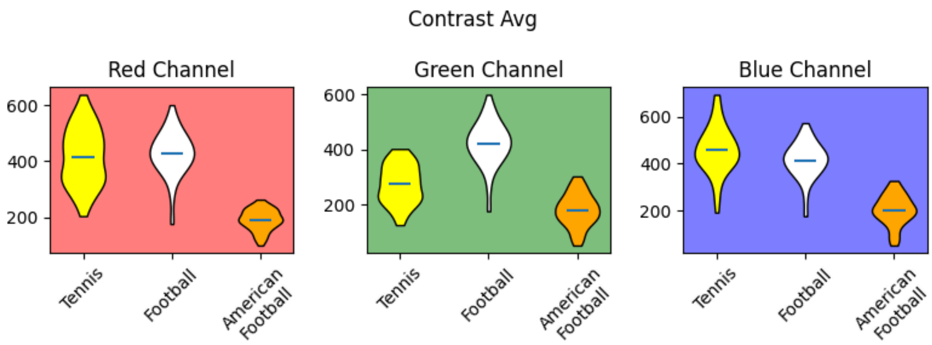

Difference 0.951

4.3 All Features

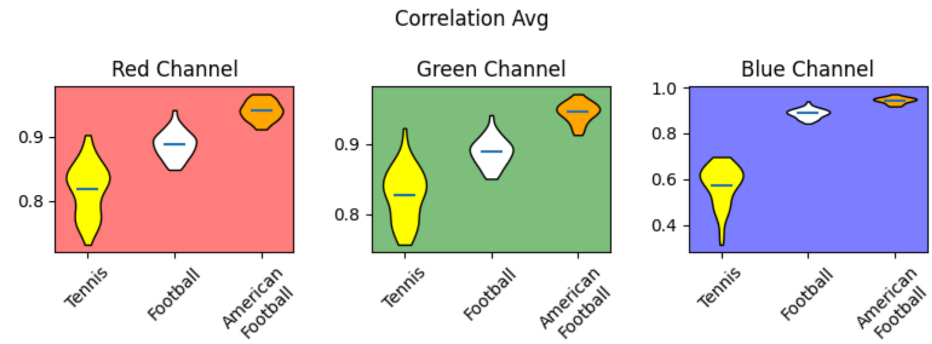




,



,



,

4.4 Code

4.4.a image_segmentation.py

```python
import os
import cv2
from cv2.typing import MatLike
import numpy as np
from segmentation.utils import fill
import math

class ImageSegmentation:
    def __init__(self, image_path: str, save_dir: str = None):
        self.processing_data = []
        self.image_path = image_path
        self.image = cv2.imread(image_path)
        self.processing_images = []
        self.save_dir = save_dir

    def log_image_processing(self, image, operation: str):
        """log the image processing"""
        self.processing_data.append(operation)
        self.processing_images.append(image)

    def gblur(self, image, ksize=(3, 3), iterations=1):
        """apply gaussian blur to the image"""
        blur = image.copy()
        for _ in range(iterations):
            blur = cv2.GaussianBlur(blur, ksize, cv2.BORDER_DEFAULT)
        self.log_image_processing(blur, f"gblur,kernel:{ksize},iterations:{iterations}")
        return blur

    def mblur(self, image, ksize=3, iterations=1):
        """apply gaussian blur to the image"""
        blur = image.copy()
        for _ in range(iterations):
            blur = cv2.medianBlur(blur, ksize)
        self.log_image_processing(
            blur, f"medianblur,kernel:{ksize},iterations:{iterations}"
        )
        return blur

    def adaptive_threshold(self, image, blockSize=15, C=3):
        """apply adaptive threshold to the image"""
        image = image.copy()
        adaptive_gaussian_threshold = cv2.adaptiveThreshold(
            src=image,
            maxValue=255,
            adaptiveMethod=cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
            thresholdType=cv2.THRESH_BINARY,
            blockSize=blockSize,
```

```python
            C=C,
        )
        self.log_image_processing(
            adaptive_gaussian_threshold,
            f"adaptive_threshold,blockSize:{blockSize},C:{C}",
        )
        return adaptive_gaussian_threshold
    def dilate(self, image, kernel=(3, 3), iterations=1, op=cv2.MORPH_ELLIPSE):
        """apply dilation to the image"""
        image = image.copy()
        kernel = cv2.getStructuringElement(op, kernel)
        dilation = cv2.dilate(
            src=image,
            kernel=kernel,
            iterations=iterations,
        )

        self.log_image_processing(
            dilation,
            f"erode,kernel:{kernel},iterations:{iterations}",
        )
        return dilation

    def erode(self, image, kernel=(3, 3), iterations=1, op=cv2.MORPH_ELLIPSE):
        """apply dilation to the image"""
        image = image.copy()
        kernel = cv2.getStructuringElement(op, kernel)
        dilation = cv2.erode(
            src=image,
            kernel=kernel,
            iterations=iterations,
        )

        self.log_image_processing(
            dilation,
            f"dilate,kernel:{kernel},iterations:{iterations}",
        )
        return dilation

    def closing(self, image, kernel=(5, 5), iterations=10):
        """apply closing to the image"""
        image = image.copy()
        kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, kernel)
        closing = cv2.morphologyEx(
            src=image,
            op=cv2.MORPH_CLOSE,
            kernel=kernel,
            iterations=iterations,
        )

        self.log_image_processing(
            closing,
            f"closing,kernel:{kernel},iterations:{iterations}",
        )
        return closing
```

```
    def opening(self, image, kernel=(5, 5), iterations=1, op=cv2.MORPH_ELLIPSE):
        """apply opening to the image"""
        image = image.copy()
        kernel = cv2.getStructuringElement(op, kernel)
        opening = cv2.morphologyEx(
            src=image,
            op=cv2.MORPH_OPEN,
            kernel=kernel,
            iterations=iterations,
        )
        self.log_image_processing(
            opening,
            f"opening,kernel:{kernel},iterations:{iterations}",
        )
        return opening

    def generic_filter(self, image, kernel, iterations=1, custom_msg="genertic_filter"):
        result = image.copy()

        for i in range(iterations):
            result = cv2.filter2D(result, -1, kernel)

        self.log_image_processing(
            result, f"{custom_msg},kernel:{kernel},iterations:{iterations}"
        )
        return result

    def dilate_and_erode(
        self, image, k_d, i_d, k_e, i_e, iterations=1, op=cv2.MORPH_ELLIPSE
    ):
        image = image.copy()
        for _ in range(iterations):
            for _ in range(i_d):
                image = self.dilate(image, (k_d, k_d), op=op)
            for _ in range(i_e):
                image = self.erode(image, (k_e, k_e), op=op)
        self.log_image_processing(
            image,
            f"dilate_and_erode,k_d:{(k_d,k_d)},i_d={i_d},k_e:{(k_e, k_e)},i_e={i_e},iterations:{iterations}",
        )
        return image

    def fill_image(self, image_data, name, show=True):
        self.log_image_processing(
            image_data[name],
            f"fill_{name}",
        )
        image_data[f"fill_{name}"] = {
            "image": fill(image_data[name]["image"].copy()),
            "show": show,
        }
```

```
    def find_ball_contours(
        self,
        image,
        circ_thresh,
        min_area=400,
        max_area=4900,
        convex_hull=False,
    ):
        img = image.copy()
        cnts = cv2.findContours(img, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
        cnts = cnts[0] if len(cnts) == 2 else cnts[1]

        blank_image = np.zeros(img.shape, dtype=img.dtype)

        for c in cnts:
            # Calculate properties
            peri = cv2.arcLength(c, True)
            # Douglas-Peucker algorithm
            approx = cv2.approxPolyDP(c, 0.0001 * peri, True)

            # applying a convex hull
            if convex_hull == True:
                c = cv2.convexHull(c)

            # get contour area
            area = cv2.contourArea(c)
            if area == 0:
                continue  # Skip to the next iteration if area is zero

            circularity = 4 * math.pi * area / (peri**2)

            if (
                (len(approx) > 5)
                and (area > min_area and area < max_area)
                and circularity > circ_thresh
            ):
                cv2.drawContours(blank_image, [c], -1, (255), cv2.FILLED)

        return blank_image


    @staticmethod
    def preprocessing(image):
        image_data = {}

        image_data["original"] = {
            "image": image.image,
            "show": True,
        }
        image_data["grayscale"] = {
            "image": cv2.cvtColor(image.image, cv2.COLOR_BGRA2GRAY),
            "show": False,
        }
```

```
        image_data["hsv"] = {
            "image": cv2.cvtColor(image.image.copy(), cv2.COLOR_BGR2HSV),
            "show": False,
        }
        (_, _, intensity) = cv2.split(image_data["hsv"]["image"])
        image_data["intensity"] = {
            "image": intensity,
            "show": False,
        }
        image_data["gblur"] = {
            "image": image.gblur(
                image_data["intensity"]["image"], ksize=(3, 3), iterations=2
            ),
            "show": False,
        }
        image_data["blur"] = {
            "image": image.mblur(
                image_data["intensity"]["image"], ksize=3, iterations=2
            ),
            "show": False,
        }

        intensity_threshold = cv2.threshold(
            image_data["intensity"]["image"], 125, 255, cv2.THRESH_BINARY
        )[1]

        image_data["intensity_threshold"] = {
            "image": intensity_threshold,
            "show": False,
        }

        name = "adap_gaus_thrsh"
        image_data[name] = {
            "image": image.adaptive_threshold(
                image=image_data["blur"]["image"].copy(),
                blockSize=19,
                C=5,
            ),
            "show": False,
        }

        image_data["open"] = {
            "image": image.opening(
                image=image_data["adap_gaus_thrsh"]["image"].copy(),
                kernel=(5, 5),
                iterations=4,
            ),
            "show": False,
        }
        image_data["dilate"] = {
            "image": image.dilate(
                image=image_data["open"]["image"].copy(),
                kernel=(3, 3),
                iterations=2,
            ),
            "show": False,
```

```
        }
        image_data["erode"] = {
            "image": image.erode(
                image=image_data["open"]["image"].copy(),
                kernel=(3, 3),
                iterations=2,
            ),
            "show": True,
        }
        fill_erode = image.fill_image(image_data, "erode")

        image_data["dilate_and_erode"] = {
            "image": image.dilate_and_erode(
                image_data["fill_erode"]["image"],
                k_d=4,
                i_d=5,
                k_e=5,
                i_e=2,
                iterations=1,
            ),
            "show": False,
        }

        contours = image.find_ball_contours(
            cv2.bitwise_not(image_data["dilate_and_erode"]["image"]),
            0.32,
        )

        image_data["contours"] = {
            "image": contours,
            "show": False,
        }

        image_data["im_1"] = {
            "image": cv2.bitwise_not(
                image_data["intensity_threshold"]["image"],
            ),
            "show": False,
        }

        image_data["im_2"] = {
            "image": cv2.bitwise_not(
                image_data["contours"]["image"],
            ),
            "show": False,
        }
        image_data["segmentation_before_recontour"] = {
            "image": cv2.bitwise_not(
                cv2.bitwise_or(
                    image_data["im_1"]["image"], image_data["im_2"]["image"]
                ),
            ),
            "show": True,
        }

        recontours = image.find_ball_contours(
```

```
            image_data["segmentation_before_recontour"]["image"],
            0.0,
            min_area=100,
            max_area=4900,
            convex_hull=True,
        )

         image_data["convex_hull"] = {
             "image": recontours,
             "show": True,
        }

        image_data["opening_after_segmentation"] = {
            "image": image.opening(
                image_data["convex_hull"]["image"],
                kernel=(3, 3),
                iterations=5,
            ),
            "show": True,
        }

        image_data["segmentation"] = {
            "image": image.find_ball_contours(
                image_data["opening_after_segmentation"]["image"],
                0.72,
                250,
                5000,
                True,
            ),
            "show": True,
        }
        return image_data
```

4.4.b utils.py

```
import os
import glob
from natsort import natsorted
import numpy as np
import matplotlib.pyplot as plt
import cv2


def get_images_and_masks_in_path(folder_path):
    images = sorted(filter(os.path.isfile, glob.glob(folder_path + "/*")))
    image_list = []
    mask_list = []
    for file_path in images:
        if "data.txt" not in file_path:
            if "GT" not in file_path:
                image_list.append(file_path)
            else:
                mask_list.append(file_path)

    return natsorted(image_list), natsorted(mask_list)


# source and modofied from https://stackoverflow.com/a/67992521
def img_is_color(img):

    if len(img.shape) == 3:
        # Check the color channels to see if they're all the same.
        c1, c2, c3 = img[:, :, 0], img[:, :, 1], img[:, :, 2]
        if (c1 == c2).all() and (c2 == c3).all():
            return True

    return False


from heapq import nlargest, nsmallest


def dice_score(processed_images, masks, save_path):
    eval = []
    score_dict = {}
    for idx, image in enumerate(processed_images):
        score = dice_similarity_score(image, masks[idx], save_path)
        score_dict[image] = score
        if len(eval) == 0 or max(eval) < score:
            max_score = score
            max_score_image = image
        if len(eval) == 0 or min(eval) > score:
            min_score = score
            min_score_image = image
        eval.append(score)
    avg_score = sum(eval) / len(eval)
    max_text = f"Max Score: {max_score} - {max_score_image}\n"
    min_text = f"Min Score: {min_score} - {min_score_image}\n"
    avg_text = f"Avg Score: {avg_score}\n"
```

```python
    print("--- " + save_path + "\n")
    print(max_text)
    print(min_text)
    print(avg_text)
    print("---")

    FiveHighest = nlargest(5, score_dict, key=score_dict.get)
    FiveLowest = nsmallest(5, score_dict, key=score_dict.get)
    with open(f"{save_path}/dice_score.txt", "w") as f:
        f.write("---\n")
        f.write(max_text)
        f.write(min_text)
        f.write(avg_text)
        f.write("---\n")
        f.write("Scores:\n")
        for idx, score in enumerate(eval):
            f.write(f"\t{score}\t{masks[idx]}\n")
        f.write("---\n")
        f.write("5 highest:\n")
        for v in FiveHighest:
            f.write(f"{v}, {score_dict[v]}\n")
        f.write("---\n")
        f.write("5 lowest:\n")
        for v in FiveLowest:
            f.write(f"{v}, {score_dict[v]}\n")

    frame_numbers = [extract_frame_number(key) for key in score_dict.keys()]

    plt.figure(figsize=(12, 3))
    plt.bar(frame_numbers, score_dict.values(), color="c")
    plt.title("Dice Score for Each Image Frame")
    plt.xlabel("Image Frame")
    plt.ylabel("Dice Similarity Similarity Score")
    plt.ylim([0.8, 1])
    plt.xticks(
        frame_numbers, rotation=90
    )  # Rotate the x-axis labels for better readability
    plt.grid(True)
    plt.tight_layout()  # Adjust the layout for better readability
    plt.savefig(f"Report/assets/dice_score_barchart.png")

    # standard deviation
    std_dev = np.std(eval)
    print(f"Standard Deviation: {std_dev}")
    mean = np.mean(eval)
    print(f"Mean: {mean}")

    # plot boxplot
    plt.figure(figsize=(12, 3))
    plt.violinplot(eval, vert=False, showmeans=True)
    plt.title("Dice Score Distribution")
    plt.xlabel("Dice Similarity Score")
    plt.grid(True)
    plt.tight_layout()
    plt.text(0.83, 0.9, f'Standard Deviation: {std_dev:.2f}',
transform=plt.gca().transAxes)
```

```
    plt.text(0.83, 0.80, f'Mean: {mean:.2f}', transform=plt.gca().transAxes)
    plt.savefig(f"Report/assets/dice_score_violin.png")

def extract_frame_number(path):
    components = path.split("/")
    filename = components[-1]
    parts = filename.split("-")
    frame_number_part = parts[-1]
    frame_number = frame_number_part.split(".")[0]
    return int(frame_number)


def dice_similarity_score(seg_path, mask_path, save_path):

    seg = cv2.threshold(cv2.imread(seg_path), 127, 255, cv2.THRESH_BINARY)[1]
    mask = cv2.threshold(cv2.imread(mask_path), 127, 255, cv2.THRESH_BINARY)[1]
    intersection = cv2.bitwise_and(seg, mask)
    dice_score = 2.0 * intersection.sum() / (seg.sum() + mask.sum())

    difference = cv2.bitwise_not(cv2.bitwise_or(cv2.bitwise_not(seg), mask))
    cv2.imwrite(save_path + f"/difference_ds_{dice_score}.jpg", difference)
    return dice_score


def show_image_list(
    image_dict: dict = {},
    list_cmaps=None,
    grid=False,
    num_cols=2,
    figsize=(20, 10),
    title_fontsize=12,
    save_path=None,
):

    list_titles, list_images = list(image_dict.keys()), list(image_dict.values())

    assert isinstance(list_images, list)
    assert len(list_images) > 0
    assert isinstance(list_images[0], np.ndarray)

    if list_titles is not None:
        assert isinstance(list_titles, list)
        assert len(list_images) == len(list_titles), "%d imgs != %d titles" % (
            len(list_images),
            len(list_titles),
        )

    if list_cmaps is not None:
        assert isinstance(list_cmaps, list)
        assert len(list_images) == len(list_cmaps), "%d imgs != %d cmaps" % (
            len(list_images),
            len(list_cmaps),
        )
```

```python
    num_images = len(list_images)
    num_cols = min(num_images, num_cols)
    num_rows = int(num_images / num_cols) + (1 if num_images % num_cols != 0 else 0)

    # Create a grid of subplots.
    fig, axes = plt.subplots(num_rows, num_cols, figsize=figsize)

    # Create list of axes for easy iteration.
    if isinstance(axes, np.ndarray):
        list_axes = list(axes.flat)
    else:
        list_axes = [axes]

    for i in range(num_images):

        img = list_images[i]
        title = list_titles[i] if list_titles is not None else "Image %d" % (i)
        cmap = (
            list_cmaps[i]
            if list_cmaps is not None
            else (None if img_is_color(img) else "gray")
        )

        list_axes[i].imshow(img, cmap=cmap)
        list_axes[i].set_title(title, fontsize=title_fontsize)
        list_axes[i].grid(grid)
        list_axes[i].axis("off")

    for i in range(num_images, len(list_axes)):
        list_axes[i].set_visible(False)

    fig.tight_layout()

    if save_path is not None:
        fig.savefig(save_path)

    plt.close(fig)


def fill(img):
    des = cv2.bitwise_not(img.copy())
    contour, hier = cv2.findContours(des, cv2.RETR_CCOMP, cv2.CHAIN_APPROX_SIMPLE)
    for cnt in contour:
        cv2.drawContours(des, [cnt], 0, 255, -1)
    return cv2.bitwise_not(des)
```

4.4.c seg_main.py

```
import os
import cv2
from tqdm import tqdm

from datetime import datetime
from segmentation.image_segmentation import ImageSegmentation
from segmentation.utils import (
    dice_score,
    get_images_and_masks_in_path,
    show_image_list,
)

import multiprocessing as mp

dir_path = os.path.dirname(os.path.realpath(__file__))
path = "data/ball_frames"


def store_image_data(log_data, time: datetime):
    """method to store in a text file the image data for processing"""
    check_path = os.path.exists(f"process_data/{time}/data.txt")
    if not check_path:
        with open(f"process_data/{time}/data.txt", "w") as f:
            for log in log_data:
                f.write(f"{log}\n")


def process_image(inputs: list[list, bool]) -> None:
    """method to process the image"""
    [image_path, save, time, save_dir] = inputs
    image = ImageSegmentation(image_path, save_dir)
    data = image.preprocessing(image)
    processed_images = {}
    for key in data.keys():
        if data[key]["show"] is not False:
            processed_images[key] = data[key]["image"]
    log_data = image.processing_data

    name = os.path.splitext(os.path.basename(image_path))[0]

    save_path = None
    if save:
        save_path = f"{save_dir}/{name}"
        if not os.path.exists(save_dir):
            os.mkdir(save_dir)
        store_image_data(log_data, time)

        if data["segmentation"]["image"] is not None:
            segmentation_path = f"{save_dir}/segmentation/"
            if not os.path.exists(segmentation_path):
                os.mkdir(segmentation_path)
            seg_path = f"{segmentation_path}{os.path.basename(image.image_path)}"
            cv2.imwrite(seg_path, data["segmentation"]["image"])
```

```python
    show_image_list(
        image_dict=processed_images,
        figsize=(10, 10),
        save_path=save_path,
    )

def process_all_images(images, save=False):
    time = datetime.now().isoformat("_", timespec="seconds")
    save_path = f"process_data/{time}"
    seg_path = f"{save_path}/segmentation"

    with mp.Pool() as pool:
        inputs = [[image, save, time, save_path] for image in images]
        list(
            tqdm(
                pool.imap_unordered(process_image, inputs, chunksize=4),
                total=len(images),
            )
        )
        pool.close()
        pool.join()

    return save_path, seg_path


def main():
    images, masks = get_images_and_masks_in_path(path)
    processed_image_path, seg_path = process_all_images(images, True)
    processed_images, _ = get_images_and_masks_in_path(seg_path)
    dice_score(processed_images, masks, seg_path)


if __name__ == "__main__":
    main()
```

4.4.d seg_main.py

```python
import os
import re
import cv2

from cv2.gapi import bitwise_and
from matplotlib import pyplot as plt
from matplotlib.artist import get

from segmentation.utils import get_images_and_masks_in_path
import numpy as np
from segmentation.utils import fill
import math
from skimage.feature import graycomatrix, graycoprops

BALL_SMALL = "Tennis"
BALL_MEDIUM = "Football"
BALL_LARGE = "American\nFootball"


def shape_features_eval(contour):
    area = cv2.contourArea(contour)

    # getting non-compactness
    perimeter = cv2.arcLength(contour, closed=True)
    non_compactness = 1 - (4 * math.pi * area) / (perimeter**2)

    # getting solidity
    convex_hull = cv2.convexHull(contour)
    convex_area = cv2.contourArea(convex_hull)
    solidity = area / convex_area

    # getting circularity
    circularity = (4 * math.pi * area) / (perimeter**2)

    # getting eccentricity
    ellipse = cv2.fitEllipse(contour)
    a = max(ellipse[1])
    b = min(ellipse[1])
    eccentricity = (1 - (b**2) / (a**2)) ** 0.5

    return {
        "non_compactness": non_compactness,
        "solidity": solidity,
        "circularity": circularity,
        "eccentricity": eccentricity,
    }


def texture_features_eval(patch):
    # # Define the co-occurrence matrix parameters
    distances = [1]
    angles = np.radians([0, 45, 90, 135])
    levels = 256
    symmetric = True
```

```
    normed = True
    glcm = graycomatrix(
        patch, distances, angles, levels, symmetric=symmetric, normed=normed
    )
    filt_glcm = glcm[1:, 1:, :, :]

    # Calculate the Haralick features
    asm = graycoprops(filt_glcm, "ASM").flatten()
    contrast = graycoprops(filt_glcm, "contrast").flatten()
    correlation = graycoprops(filt_glcm, "correlation").flatten()

    # Calculate the feature average and range across the 4 orientations
    asm_avg = np.mean(asm)
    contrast_avg = np.mean(contrast)
    correlation_avg = np.mean(correlation)
    asm_range = np.ptp(asm)
    contrast_range = np.ptp(contrast)
    correlation_range = np.ptp(correlation)

    return {
        "asm": asm,
        "contrast": contrast,
        "correlation": correlation,
        "asm_avg": asm_avg,
        "contrast_avg": contrast_avg,
        "correlation_avg": correlation_avg,
        "asm_range": asm_range,
        "contrast_range": contrast_range,
        "correlation_range": correlation_range,
    }


def initialise_channels_features():
    def initialise_channel_texture_features():
        return {
            "asm": [],
            "contrast": [],
            "correlation": [],
            "asm_avg": [],
            "contrast_avg": [],
            "correlation_avg": [],
            "asm_range": [],
            "contrast_range": [],
            "correlation_range": [],
        }

    return {
        "blue": initialise_channel_texture_features(),
        "green": initialise_channel_texture_features(),
        "red": initialise_channel_texture_features(),
    }

def initialise_shape_features():
    return {
        "non_compactness": [],
        "solidity": [],
```

```
        "circularity": [],
        "eccentricity": [],
    }


def get_all_features_balls(path):
    features = {
        BALL_LARGE: {
            "shape_features": initialise_shape_features(),
            "texture_features": initialise_channels_features(),
        },
        BALL_MEDIUM: {
            "shape_features": initialise_shape_features(),
            "texture_features": initialise_channels_features(),
        },
        BALL_SMALL: {
            "shape_features": initialise_shape_features(),
            "texture_features": initialise_channels_features(),
        },
    }

    images, masks = get_images_and_masks_in_path(path)
    for idx, _ in enumerate(images):
        image = images[idx]
        mask = masks[idx]
        msk = cv2.imread(mask, cv2.IMREAD_GRAYSCALE)
        _, msk = cv2.threshold(msk, 127, 255, cv2.THRESH_BINARY)

        # overlay binay image over it's rgb counterpart
        img = cv2.imread(image)
        img = cv2.bitwise_and(cv2.cvtColor(msk, cv2.COLOR_GRAY2BGR), img)
        contours, _ = cv2.findContours(msk, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)

        for contour in contours:
            area = cv2.contourArea(contour)
            ball_img = np.zeros(msk.shape, dtype=np.uint8)
            cv2.drawContours(ball_img, contour, -1, (255, 255, 255), -1)
            fill_img = cv2.bitwise_not(fill(cv2.bitwise_not(ball_img)))
            rgb_fill = cv2.bitwise_and(cv2.cvtColor(fill_img, cv2.COLOR_GRAY2BGR), img)

            out = fill_img.copy()
            out_colour = rgb_fill.copy()

            # Now crop image to ball size
            (y, x) = np.where(fill_img == 255)
            (topy, topx) = (np.min(y), np.min(x))
            (bottomy, bottomx) = (np.max(y), np.max(x))
            padding = 3
            out = out[
                topy - padding : bottomy + padding, topx - padding : bottomx + padding
            ]
            out_colour = out_colour[
```

```
                topy - padding : bottomy + padding, topx - padding : bottomx + padding
            ]

            # getting ball features
            shape_features = shape_features_eval(contour)
            texture_features_colour = {
                "blue": texture_features_eval(out_colour[:, :, 0]),
                "green": texture_features_eval(out_colour[:, :, 1]),
                "red": texture_features_eval(out_colour[:, :, 2]),
            }

            # segmenting ball by using area
            if area > 1300:  # football
                append_ball = BALL_LARGE
            elif area > 500:  # soccer_ball
                append_ball = BALL_MEDIUM
            else:  # tennis ball
                append_ball = BALL_SMALL

            for key in shape_features:
                features[append_ball]["shape_features"][key].append(shape_features[key])

            for colour in texture_features_colour.keys():
                for colour_feature in texture_features_colour[colour]:
                    features[append_ball]["texture_features"][colour][
                        colour_feature
                    ].append(texture_features_colour[colour][colour_feature])
    return features


def feature_stats(features, ball, colours=["blue", "green", "red"]):
    def get_stats(array):
        return {
            "mean": np.mean(array),
            "std": np.std(array),
            "min": np.min(array),
            "max": np.max(array),
        }

    def get_ball_shape_stats(features, ball):
        feature_find = ["non_compactness", "solidity", "circularity", "eccentricity"]
        return {
            feature: get_stats(features[ball]["shape_features"][feature])
            for feature in feature_find
        }
    def get_ball_texture_stats(features, ball, colour):
        feature_find = ["asm_avg", "contrast_avg", "correlation_avg"]
        return {
            texture: get_stats(features[ball]["texture_features"][colour][texture])
            for texture in feature_find
        }

```

```python
    stats = {
        ball: {
            "shape_features": get_ball_shape_stats(features, ball),
            "texture_features": {
                colour: get_ball_texture_stats(features, ball, colour)
                for colour in colours
            },
        },
    }
    return stats


def get_histogram(data, Title):
    """
    data {ball: values}
    """
    for ball, values in data.items():
        plt.figure(figsize=(3,3))
        plt.hist(values, bins=20, alpha=0.5, label=ball)
        plt.xlabel(Title)
        plt.ylabel("Frequency")
        plt.legend()
        plt.tight_layout()
        plt.savefig("Report/assets/features/"+ Title + "_histogram_" +
ball.replace("\n", "_"))
    # plt.show()


if __name__ == "__main__":
    features = get_all_features_balls("data/ball_frames")

    balls = [
        BALL_SMALL,
        BALL_MEDIUM,
        BALL_LARGE,
    ]

    non_compactness = {
        ball: features[ball]["shape_features"]["non_compactness"] for ball in
balls
    }
    solidity = {ball: features[ball]["shape_features"]["solidity"] for ball in
balls}
    circularity = {
        ball: features[ball]["shape_features"]["circularity"] for ball in
balls
    }
    eccentricity = {
        ball: features[ball]["shape_features"]["eccentricity"] for ball in
balls
    }

    get_histogram(non_compactness, "Non-Compactness")
    get_histogram(solidity, "Soliditiy")
    get_histogram(circularity, "Circularity")
```

```python
get_histogram(eccentricity, "Eccentricity")

channel_colours = ["red", "green", "blue"]

def get_ch_features(feature_name):
    return {
        colour: {
            ball: features[ball]["texture_features"][colour][feature_name]
            for ball in balls
        }
        for colour in channel_colours
    }

def get_ch_stats(feature_data, colours=channel_colours):
    return [[feature_data[colour][ball] for ball in balls] for colour in colours]

asm_avg = get_ch_features("asm_avg")
contrast_avg = get_ch_features("contrast_avg")
correlation_avg = get_ch_features("correlation_avg")
asm_range = get_ch_features("asm_range")

asm_data = get_ch_stats(asm_avg)
contrast_data = get_ch_stats(contrast_avg)
correlation_data = get_ch_stats(correlation_avg)
asm_range_data = get_ch_stats(asm_range)

asm_title = "ASM Avg"
contrast_title = "Contrast Avg"
correlation_title = "Correlation Avg"
asm_range_title = "ASM Range Avg"

plt_colours = ["yellow", "white", "orange"]
channels = ["Red Channel", "Green Channel", "Blue Channel"]

plt.figure()

def get_boxplot(data, title, colours=plt_colours, rows=3, columns=3, offset=0):
    channels = ["Red Channel", "Green Channel", "Blue Channel"]

    fig = plt.figure(figsize=(8,3))  # Get the Figure object
    fig.suptitle(title)  # Set the overall title
    for i, d in enumerate(data):
        ax = plt.subplot(rows, columns, i + offset + 1)
        ax.set_facecolor(channel_colours[i])
        ax.patch.set_alpha(0.5)
        violins = plt.violinplot(
            d, showmeans=True, showmedians=False, showextrema=False
        )
        for j, pc in enumerate(violins["bodies"]):
            pc.set_facecolor(colours[j])
            pc.set_edgecolor("black")
            pc.set_alpha(0.2)
        plt.xticks([1, 2, 3], balls, rotation=45)
        plt.title(channels[i])
```

```python

def get_boxplot_specific(data, title, i, colours=plt_colours):

    plt.figure(figsize=(2.5,6))
    d = data[i]
    violins = plt.violinplot(
        d, showmeans=True, showmedians=False, showextrema=False
    )
    for j, pc in enumerate(violins["bodies"]):
        pc.set_facecolor(colours[j])
        pc.set_edgecolor("black")
        pc.set_alpha(0.5)
    plt.xticks([1, 2, 3], balls, rotation=45)
    plt.title(title + '\n' + channels[i])
    ax = plt.gca()  # Get the current Axes instance
    ax.set_facecolor(channel_colours[i])  # Set the background color
    ax.patch.set_alpha(0.1)  # Set the alpha value

columns = 3
rows = 1

get_boxplot_specific(asm_data, asm_title, 2)
plt.tight_layout()
plt.savefig("Report/assets/features/asm_data_blue_channel")
plt.close()

get_boxplot_specific(asm_range_data, asm_range_title, 2)
plt.tight_layout()
plt.savefig("Report/assets/features/asm_range_data_blue_channel")
plt.close()

get_boxplot_specific(contrast_data, contrast_title, 0)
plt.tight_layout()
plt.savefig("Report/assets/features/contrast_data_red_channel")
plt.close()

get_boxplot_specific(correlation_data, correlation_title, 1)
plt.tight_layout()
plt.savefig("Report/assets/features/correlation_green_channel")
plt.close()
```

4.4.e tracking_main.py

```
from matplotlib import pyplot as plt
import numpy as np


def kalman_predict(x, P, F, Q):
    xp = F * x
    Pp = F * P * F.T + Q
    return xp, Pp


def kalman_update(x, P, H, R, z):
    S = H * P * H.T + R
    K = P * H.T * np.linalg.inv(S)
    zp = H * x

    xe = x + K * (z - zp)
    Pe = P - K * H * P
    return xe, Pe


def kalman_tracking(
    z,
    x01=0.0,
    x02=0.0,
    x03=0.0,
    x04=0.0,
    dt=0.5,
    nx=0.16,
    ny=0.36,
    nvx=0.16,
    nvy=0.36,
    nu=0.25,
    nv=0.25,
    kq=1,
    kr=1,
):
    # Constant Velocity
    F = np.matrix([[1, dt, 0, 0], [0, 1, 0, 0], [0, 0, 1, dt], [0, 0, 0, 1]])

    # Cartesian observation model
    H = np.matrix([[1, 0, 0, 0], [0, 0, 1, 0]])

    # Motion Noise Model
    Q = kq*np.matrix([[nx, 0, 0, 0], [0, nvx, 0, 0], [0, 0, ny, 0], [0, 0, 0,
nvy]])
    # Measurement Noise Model
    R = kr*np.matrix([[nu, 0], [0, nv]])

    x = np.matrix([x01, x02, x03, x04]).T
    P = Q

    N = len(z[0])
    s = np.zeros((4, N))

```

```python
    for i in range(N):
        xp, Pp = kalman_predict(x, P, F, Q)
        x, P = kalman_update(xp, Pp, H, R, z[:, i])
        val = np.array(x[:2, :2]).flatten()
        s[:, i] = val

    px = s[0, :]
    py = s[1, :]

    return px, py


def rms(x, y, px, py):
    return np.sqrt(1/len(px)  * (np.sum((x - px)**2 + (y - py)**2)))

def mean(x, y, px, py):
    return np.mean(np.sqrt((x - px)**2 + (y - py)**2))

if __name__ == "__main__":

    x = np.genfromtxt("data/x.csv", delimiter=",")
    y = np.genfromtxt("data/y.csv", delimiter=",")
    na = np.genfromtxt("data/na.csv", delimiter=",")
    nb = np.genfromtxt("data/nb.csv", delimiter=",")
    z = np.stack((na, nb))

    dt = 0.5
    nx = 160.0
    ny = 0.00036
    nvx = 0.00016
    nvy = 0.00036
    nu = 0.00025
    nv = 0.00025

    px1, py1 = kalman_tracking(z=z)

    nx = 0.16 * 10
    ny = 0.36
    nvx = 0.16 * 0.0175
    nvy = 0.36 * 0.0175
    nu = 0.25
    nv = 0.25 * 0.001
    kq = 0.0175
    kr = 0.0015
    px2, py2 = kalman_tracking(
        nx=nx,
        ny=ny,
        nvx=nvx,
        nvy=nvy,
        nu=nu,
        nv=nv,
        kq=kq,
        kr=kr,
        z=z,
    )

```

```
plt.figure(figsize=(12, 5))

plt.plot(x, y, label='trajectory')
plt.plot(px1, py1, label=f'intial prediction, rms={round(rms(x, y, px1, py1), 3)}')
print(f'initial rms={round(rms(x, y, px1, py1), 3)}, mean={round(mean(x, y, px1, py1), 3)}')
plt.plot(px2, py2,label=f'optimised prediction, rms={round(rms(x, y, px2, py2), 3)}')
print(f'optimised rms={round(rms(x, y, px2, py2), 3)}, mean={round(mean(x, y, px2, py2), 3)}')
plt.scatter(na, nb,marker='x',c='k',label=f'noisy data, rms={round(rms(x, y, na, nb), 3)}')
print(f'noise rms={round(rms(x, y, na, nb), 3)}, mean={round(mean(x, y, na, nb), 3)}')
plt.legend()

plt.title("Kalman Filter")
plt.savefig("Report/assets/tracking/kalman_filter.png")
# plt.show()
```