Image Analysis and Computer Vision

Professor Vincenzo Caglioti

Visual Analysis of Sport Events: Tennis

(Full Project)

2nd Semester – 2023-2024 A.Y.

Paolo Riva: 10938975

e-mail: paolo13.riva@mail.polimi.it

Michelangelo Stasi: 10787871

e-mail: michelangelo.stasi@mail.polimi.it

Mihai-Viorel Grecu: 10879851

e-mail: mihaiviorel.grecu@mail.polimi.it

Index

1 Introduction

Data collection in sports and its successful analysis has been a keystone in improving the performance of individual players and of entire teams over the past decades. Additionally, the statistics obtained from the data analysis have proven to have a significant positive economic impact, thus becoming of crucial interest in our contemporary era.

As such, to improve data analysis and quality of statistics in sport events, a growing interest in sports data to be used in visualization research has been observed so that sports visualization to offer new approaches to exploring, making sense of, and communicating sports data. As visualizations can be more accessible and more meaningful than traditional statistical analysis the number of visualizations of sports data has grown rapidly over the past decades.

With the addition of better illustrations of sports data, better analysis can be performed to improve safety, performance and to attract the public with interesting and appealing statistics.

This project is publicly available at: https://github.com/MyKe01/IACV-project Instructions to run the program can be found at such link in the README.md file.

1.1 Goal of the project

Ranking 4th in the global ranking of most popular sports with 1 billion fans, Tennis, has had a continuous rise in popularity in the last years, with a 43% participation increase in adult players compared to 2021 in the UK, as well as a staggering 49% total U.S. participation increase in Q3 of 2020 vs Q3 of 2019 (Tennis Industry Association).

As a result of the beforementioned statistics, the goal of our project is to compute the trajectory of a tennis player through a monocular video taken by a single static camera. This translates in using the main fixed camera which films the tennis court from behind one of the tennis players as exampled in *Figure 1.1*.

To achieve the desired outcome, the problem was separated into several intermediate steps described below:

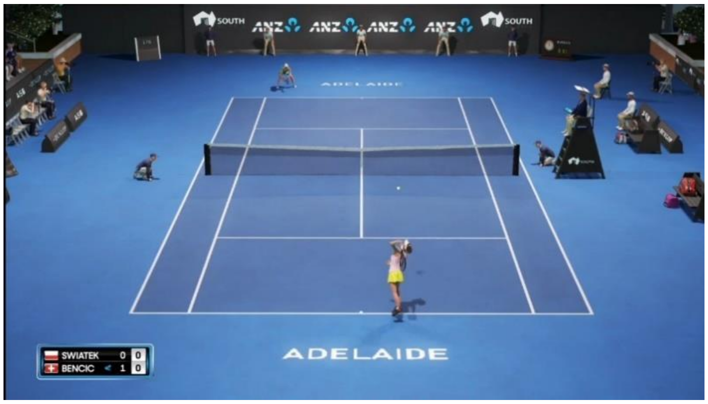


Figure 1.1 Picture over the tennis court from a monocular video taken by a single static camera

1. Identify the homography from the field to the image;
2. Use of the Human Pose Estimation method (Deep Learning model based) to identify the articulated segments of the players;
3. Checking the dynamics of the feet of the players (static/moving) by checking whether the position of the feet (on the tennis field plane) is constant along a short time interval (feet are static if are placed on the ground);
4. Collecting the time-sequence of the step points (i.e. the static position if the feet).

As a final challenge, racket hits and their respective time instants are selected to then allow the evaluation of statistics during the ball exchange considered between the players.

1.2 State of the art and our choice

The field of sports analysist through broadcast-available videos has seen a global increase in relevance and market appeal in the last decade, also helped by the recent steps achieved in the fields of Machine Learning, Deep Learning and their applications to computer vision solutions. This allows many implementations to be selectable regarding different hardware and software performance requirements.

Researching state of the art methods in consumer-level machines is the first step in the process of finding the correct approach to implementing the algorithm in a correct way.

1.2.1 Homography Computation

Analysis in tennis broadcast videos requires being able to detect the field plane and its orientation with respect to the camera. This can be achieved using OpenCV, a powerful computer vision library. OpenCV has become the standard for computer vision analysis in Python environment: OpenCV's cv2 module provides a function called findHomography, which computes the homography matrix. This matrix maps points from the field plane in the video to a reference plane, allowing for accurate determination of the court's position and orientation. By identifying corresponding points in the video frame and the reference image, findHomography can compute the transformation needed to align the field with the camera view.

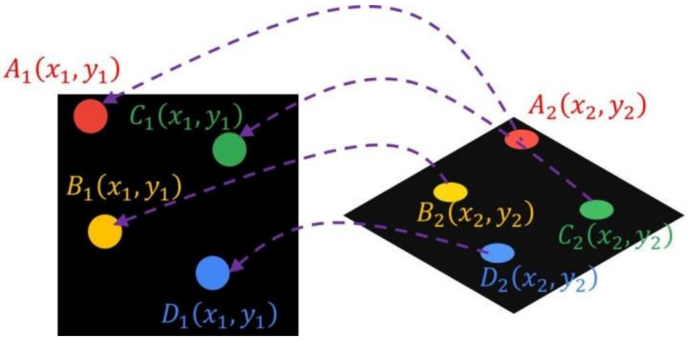


Figure 1.2 Points correlation from projection to plane using homography

1.2.2 Human Pose Estimation

Regarding human pose estimation, OpenPose is widely recognized as one of the pioneering models in multi-person pose estimation, using a multi-stage CNN (Convolutional Neural Networks) to predict keypoints and part affinity fields (PAFs).

While not being the top-ranking model (11th place in 2D Human Pose Estimation on COCO-WholeBody), having still very good efficiency compared to the top models and being an older solution which allows for more documentation and implementations, OpenPose has been chosen as the initial solution for the Human Pose Estimation task. However, since in preliminary implementation was performing excessively slow, real-time Human Pose Estimation using MediaPipe platform has been chosen as the solution to be used. This has been our choice due to being faster in our testing (due to predicting the landmarks with a single shot approach) and more accurate (due to using pose refinement and temporal filtering to smooth out the jitter and noise), an open-source platform maintained by Google with a comprehensive set of pre-trained models that make applications for tasks like pose estimation, especially in sports, easier with the real-time feedback on form and posture (being a balance between speed and accuracy).

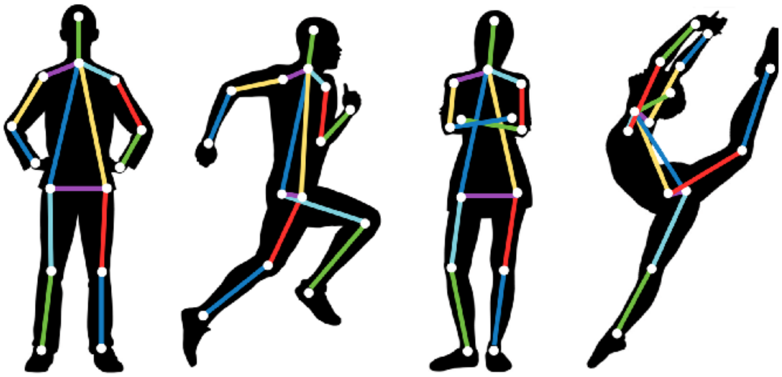


Figure 1.3 Human pose estimation - landmarks of the human body

1.2.3 Ball Detection and Object Classification

Within the tennis ball tracking domain, YOLOv8 has been our first choice for the object detection model since being the newer version of YOLOv7 which stays at the base of the best available tracker (SMILEtrack according to MOT17 and MOT20 benchmarks). However, we had to look for a more accurate object tracker suited for tiny and fast-moving objects. Thus, TrackNet model has developed as the state-of-the-art to also be implemented in our algorithm since being developed for sport analytics, specifically for real-time tracking of fast-moving tennis balls. It shows higher accuracy, robustness and speed than other object tracking models which have a broader tracking scenario, being more versatile rather than performant in tennis ball tracking.

2 Our approach

2.1 Homography computation through OpenCV

The first step is to compute the homography from the image to the field using the lines of the real field, i.e., the real dimensions of the field length and width (respectively 23.78 m and 10.97 m).

To do this, in a first moment we used an approach seen during our laboratory classes, in which we select the vertices of the field in the image and map them to the real vertices of the field.

The only problem with this approach is that it is not automatic; to address this problem, we decided to implement an automatic field lines detection and then map the intersections of these lines with the vertices of the real vertices of the field.

To do this, we used the *HoughLinesP* function to find the lines with most intersections; in particular, in the Hough Transform, you can see that even for a line with two arguments, it takes a lot of computation. Probabilistic Hough Transform is an optimization of the Hough Transform. It doesn't take all the points into consideration. Instead, it takes only a random subset of points which is sufficient for line detection. The Hough Transform is a popular technique to detect any shape, if you can represent that shape in a mathematical form. It can detect the shape even if it is broken or distorted a little bit. It is implemented by the library OpenCV.

Once we detected the lines through Probabilistic Hough Transform, for each line we find, the intersections with the other lines and the lines with most intersections get a fill mask command; in this way, the lines detected that are the boundaries of a rectangular area are now a single line (see example in Figure 2.1).

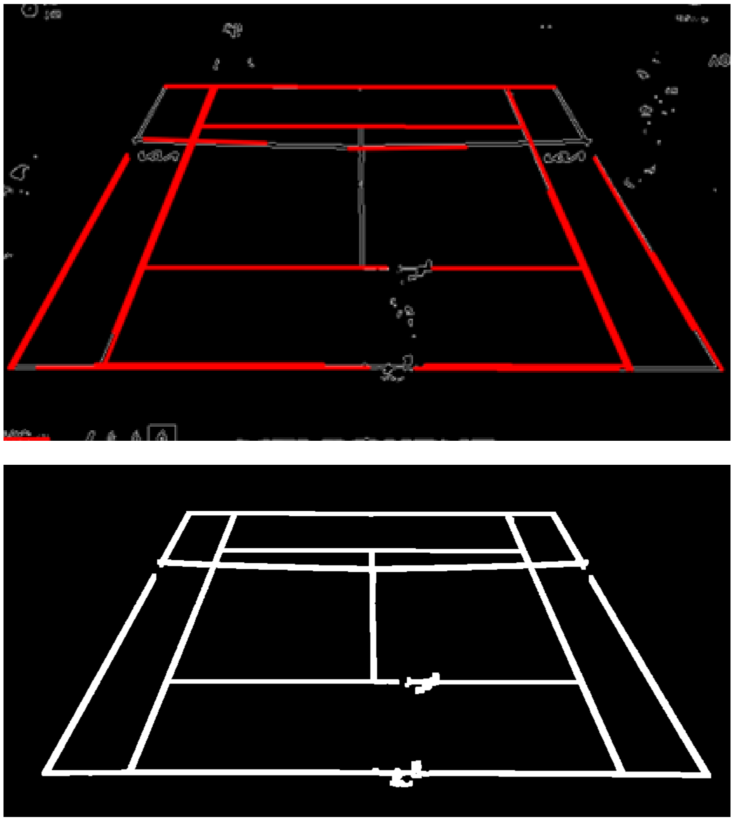


Figure 2.1 Example for Fill mask usage - region suitable for dilation

Now we apply again the Hough Transform on the same image but modified with the dilated lines and we look for the vertices of the field.

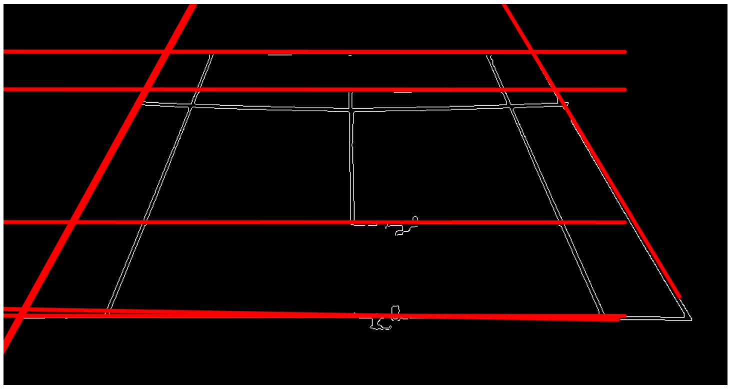


Figure 2.2 Vertices computation through the Hough Transform

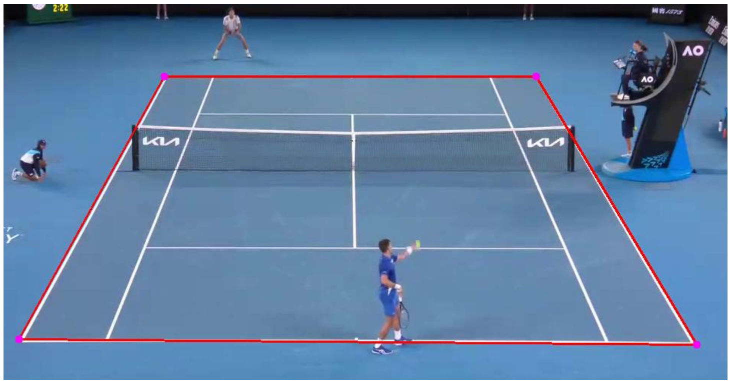


Figure 2.3 Tennis field exterior lines identification

Once we have the correct field vertices, we apply the homography as previously said. We also compute the ratio coefficient to get the equivalent of 1 pixel in meters in our video; it will be fundamental later.

2.2 Media Pipe Human Detection

To apply the Human Pose Estimation, an important issue regarding the algorithm had to be considered, this being the computational effort.

Taking as reference the multiple threads on StackOverflow regarding the experience of multiple programmers implementing Human pose estimation, as well as our experience, it was observed that multiple object detection was possible in a single video. However, the computational speed was significantly slower compared to the Human Pose Estimation of a single player. Thus, to address this problem, multithreading was implemented, and a specific region of the frame video was assigned to each thread. In this way, less detections with less noise were found, as well as avoiding including the other humans present inside the video.

As a result, to find the specific region autonomously, it was decided to implement a Human Detection algorithm since it is less effort demanding that the Human Pose Estimation if there are multiple humans in the video frame. Therefore, a pre-trained model was implemented through

MediaPipe library called *EfficientDet_Lite0*, being a lite version of the object detection models EfficientDets [1].

The choice of using *EfficientDet_Lite0* model was based on multiple factors such as:

- Being a recommended object detection model by Google Ai Edge ;
- Having tested and proven that its accuracy in tennis players detection compared to other models such as *SSD MobileNetV2*, and the other versions of the EfficientDet_Lite model (From 1 to 4, see Table 1) is better;
- Having the fastest computational time between the models (see Table 1);
- Being less resource-intensive in terms of both memory and processing power.

Table 1 List of mobile-size lite EfficientDet models

Model	mAP (float)	Quantized mAP (int8)	Parameters	Mobile latency
EfficientDet-lite0	26.41	26.10	3.2M	36ms
EfficientDet-lite1	31.50	31.12	4.2M	49ms
EfficientDet-lite2	35.06	34.69	5.3M	69ms
EfficientDet-lite3	38.77	38.42	8.4M	116ms
EfficientDet-lite3x	42.64	41.87	9.3M	208ms
EfficientDet-lite4	43.18	42.83	15.1M	260ms

mAP is the mean Average Precision on the COCO 2017 dataset, which is calculated according to the following formula:

$$mAP = \frac{1}{n}\sum_{k=1}^{n} AP_k$$

AP_k= Average Precision of class k

n = the number of classes

Within this equation, we begin by determining distinct precisions for each class by applying various IoU (Intersection over Union) thresholds. This involves categorizing a detection as a true positive when the IoU metric exceeds the specified threshold between the predicted box and the actual box. Once we have computed these precisions for a particular class, we calculate their average and repeat the process for other classes. Ultimately, we derive the mean of these Average Precisions across all classes, yielding the mAP value for the respective model.
The idea is that we are detecting all humans present in the frame and filter them based on the distance from the center of the video with respect to its width; in fact, the closest humans to the center will be without any doubt the players of the match.

The next step is to understand who the player at the top of the frame is and who is the one at the bottom; to determine the position of the players, our solution is to look at who is below or above the center of the frame with respect to its height.

Another problem found during implementation was the noise of the detection. In fact, sometimes the players were not detected correctly, and the detection moved far away from the previous detection. To address this problem, our idea was to store the previous detection if the new detection was too far away from it; when the distance is below a certain threshold, the previous detection is updated with the new one.

Finally, to retrieve from these detections the size of the frame in which we had to apply the Human Pose Estimation, we saved the extreme points reached during the detection; in this way we have a rectangular area in which only the player considered will move during the whole video. Each player has its own area.

2.3 Human Pose Estimation

Once we got the cropped part of the frame for each player, we create two different threads at each frame that apply the pose model to the corresponding cropped part of the frame. By isolating the computation to a specific portion of the frame, it is faster and more accurate.

2.3.1 Human Pose Estimation using OpenPose

During the early phase of the project, we decided to test a pre-trained model b an external research team, based on the OpenPose model, that trained the human pose detection and estimation through generic human images. The choice was taken since, for the knowledge of the state of the art in the early phase of the project, such team had implemented the human pose estimation method in a straight-forward, out-of-the-box ready approach. On the other hand, in terms of computational time and noise seen on the output, the model was performing poorly especially in high noise – high movement scenarios, which made the implementation not the most suitable for our case (Figure 2.4).

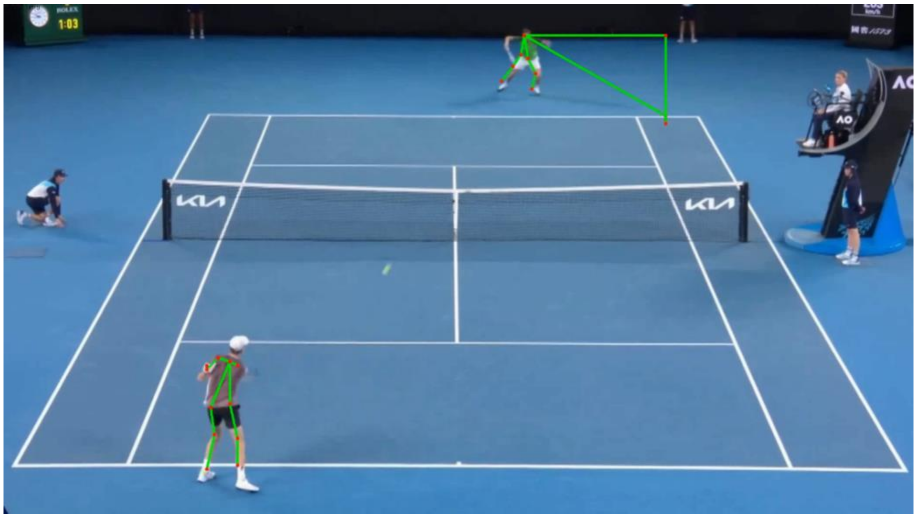


Figure 2.4 Human Pose Estimation through OpenCV pre-trained model results

For this reason, after attempting some tweaking on the available parameters, we decided to search for other, well-known and better performing implementations.

2.3.2 Human Pose Estimation using MediaPipe

Checking the most implemented models throughout papers related to the topic, as well as external projects, we found out that the module “MediaPipe” had a built-in Human Pose model that was significantly more accurate and faster than the OpenPose one, even if its complexity was higher.

Thus, by looking at the documentation, we retrieved from the computed pose the feet to detect when the players were moving or not; we also used this information to plot the live position in the rectified image and the live trajectory for each player.

To determine whether a player was moving or not, we check if the two feet are static or not.

To do this, at each frame for each foot we check if the previous position is at a distance greater than a certain threshold; if it is, then the foot is moving, otherwise it is static. First, we computed the position of the feet in the rectified image, since in this way we can treat both players in the same way with the same threshold, otherwise the top player would need a threshold smaller than the one of the bottom player, due to perspective.

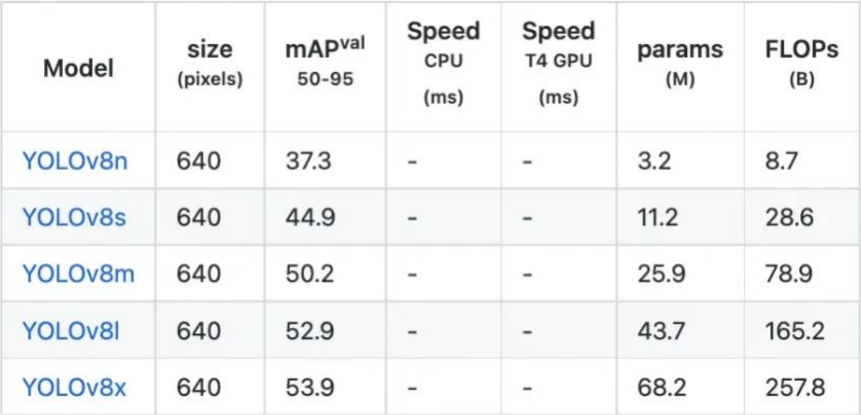
The problem with this was about deciding when to update the previous position, since if it is updated at each frame, even if a foot is moving, the position will change for several pixels for each frame comparable to the noise in case of static foot and therefore the foot would be detected static even if it is not.

We addressed this problem by updating the previous feet positions every 5 frames and by using a threshold of 5 pixels; if in 5 frames the foot has moved more than 5 pixels, then it is moving, otherwise it is static.

2.4 Ball trajectory detection

Being able to detect the ball is presented as a hard task since it is highly sensitive to framerate, speed and pixel motion blur.
The first choice we tested, resulting as the most used candidate from the state of the art, was the “You Only Look Once (YOLO”) method, specifically in the version “YOLOv8x”. This version was chosen being the most powerful and accurate available at the time, due to the number of parameters it was trained on, and since the project didn’t aim at achieving low processing time per frame in the first place. YOLOv8 can detect tennis balls labeling them as “Sports ball”, but the performance showed to be very poor straight after the first tests. This was justified by the fact that all main YOLO models are not intended to be precise on fast-moving objects, de facto not suiting our scenario very well (Table 2).

Table 2 YOLOv8 model performance comparison

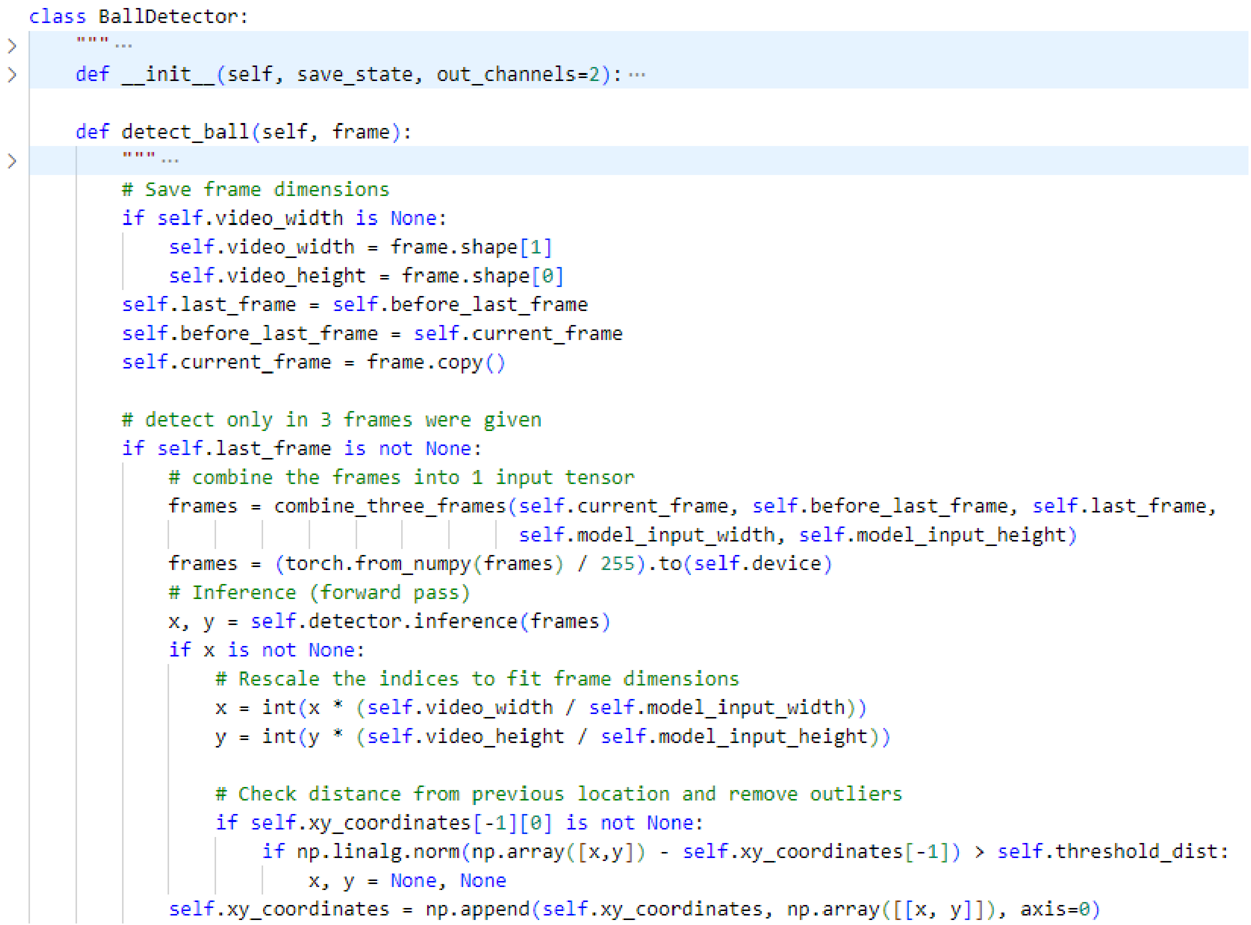


Model	size (pixels)	mAP^{val} 50-95	Speed CPU (ms)	Speed T4 GPU (ms)	params (M)	FLOPs (B)
YOLOv8n	640	37.3	-	-	3.2	8.7
YOLOv8s	640	44.9	-	-	11.2	28.6
YOLOv8m	640	50.2	-	-	25.9	78.9
YOLOv8l	640	52.9	-	-	43.7	165.2
YOLOv8x	640	53.9	-	-	68.2	257.8

Thus, a better solution in detecting the tennis ball during exchanges between the tennis players was found to be “Tracknet” [2] , a tennis ball tracking from broadcast video by deep learning networks. The precision, recall, and F1-measure of TrackNet reach 99.7%, 97.3%, and 98.5%, respectively. To prevent overfitting, 9 additional videos are partially labeled together with a subset from the previous dataset to implement 10-fold cross-validation, and the precision, recall, and F1-measure are 95.3%, 75.7%, and 84.3%, respectively.

TrackNet takes multiple consecutive frames as input, so that the model learns not only object tracking but also trajectory to enhance its capability of positioning and recognition. TrackNet will generate a gaussian heat map centered on the ball to indicate the position of the ball.

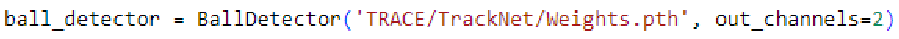
Such solution was ported to many known tennis analyzers to be the most suitable code implementation. By checking the available implementations, we found “TRACE”, a tennis analyzer created by Harsh Gupta, to be able to provide a high-level interface and code implementation to process the clip through TrackNet.



```
class BallDetector:
    """...
    def __init__(self, save_state, out_channels=2):...

    def detect_ball(self, frame):
        """...
        # Save frame dimensions
        if self.video_width is None:
            self.video_width = frame.shape[1]
            self.video_height = frame.shape[0]
        self.last_frame = self.before_last_frame
        self.before_last_frame = self.current_frame
        self.current_frame = frame.copy()

        # detect only in 3 frames were given
        if self.last_frame is not None:
            # combine the frames into 1 input tensor
            frames = combine_three_frames(self.current_frame, self.before_last_frame, self.last_frame,
                                          self.model_input_width, self.model_input_height)
            frames = (torch.from_numpy(frames) / 255).to(self.device)
            # Inference (forward pass)
            x, y = self.detector.inference(frames)
            if x is not None:
                # Rescale the indices to fit frame dimensions
                x = int(x * (self.video_width / self.model_input_width))
                y = int(y * (self.video_height / self.model_input_height))

                # Check distance from previous location and remove outliers
                if self.xy_coordinates[-1][0] is not None:
                    if np.linalg.norm(np.array([x,y]) - self.xy_coordinates[-1]) > self.threshold_dist:
                        x, y = None, None
            self.xy_coordinates = np.append(self.xy_coordinates, np.array([[x, y]]), axis=0)
```

TRACE provides a class “BallDetector” with a method *detect_ball* that directly loads the pre-trained TrackNet model and does the inference through *torch* on the frame passed as argument.

Importing BallDetector and processing the clip through it allowed us to directly spot the performance and precision improvement with respect to YOLOv8, specifically when the ball was close to the net during points.



```
ball_detector = BallDetector('TRACE/TrackNet/Weights.pth', out_channels=2)
```

Even with the implementation of TrackNet, the main issue came out to be processing ball position in frames where it was close to a player or hit by him, due to the probability associated with ball detection being high when referring to hands-feet of the player. For this, reason, missing several frames of information or having sudden changes in ball detection due to the wrongly-associations of the ball to player body parts was fixed in two ways by:

- Implementation of a *processBallTrajectory* function, in charge of detecting the ball through the *ball_detector* object and discarding noisy positions not related to the past observed state of the ball;
- Implementation of an *interpolate_missing_values* function that interpolates the known trajectory through a spline interpolation method considering a dynamic processing window to reconstruct the behavior of the ball close to players and during shots.

Applying both functions on the clip allowed to gather a good trajectory estimation of the ball just from the video itself, with low noise on ball detection and in unknown position.

Ball positions detected by TrackNet are shown in green on each frame, while interpolation results are visible in red.

2.5 Statistics computed

The last task required by the project specifications was to perform an attempt on detecting racket hits and compute statistics between them.

2.5.1 Racket hits

Performing racket hit detection was thought considering the trajectory of the ball and the gradient change in trajectory happening. Since a racket hit by a player reverts the ball movement from top to bottom (or vice versa) considering any conventional tennis camera placement, we first implemented the algorithm to mark any frame where a vertical gradient sign inversion happened.

Due to the noise on ball position, related to subsequent frames, having the vertical coordinate oscillate (especially with a high framerate) caused the vertical derivative to change sign much more frequently than expected. This, along with frames where specific ball bounces happened to be marked as well due to ball bouncing in perspective, causing the vertical coordinate to change derivative as well, caused a lot of vertical derivative sign changes unrelated to racket hits by the players.

To solve this, a quadratic Savitzky-Golay smoothing filter has been applied to the ball trajectory with a sufficiently large window to smooth the data and be able to filter out any noise on ball detection. The smoothing window has been tweaked to filter out ball bounces closer to the net as well, without compromising quick ball exchanges between players. The optimal window was found to be around 90% of the frame per second count through empiric tests.

The resulting ball position to analyze has come out to be including the trajectory change we would expect from a 2-dimensions view of the pitch from the top in terms of velocity sign change caused by racket hits, leaving out basically only the cases where the ball bounced very high in perspective or very close to a player, de-facto switching the vertical derivative in sign away from an actual hit or just before the actual hit (e.g. a high hit from the top player bouncing close to the bottom player caused the ball to start moving up in the original view, to then proceed vertically at a speed with higher module after the hit by the bottom player).

These last cases were presented as a synchronization issue: while the vertical derivative changed sign as expected, in the perspective this happened before the actual racket hit in isolated cases, so a first synchronization condition, specifically to consider only the gradient changes within a certain distance from a player (chosen as a multiple of the player's detected height) was imposed to ignore residual gradient changes from the previous filtering not related to racket hits but exclusively to far-away bounces.

The implementation was then improved by considering a frame interval around each vertical gradient change in order to handle gradient changes close to the player due to high bounces (which was the second left-out case), and to assign the final racket hit to the frame where the average hand velocity was the highest among both players' hands, since the racket hit always caused an increase in hand speed to be performed. This has been achieved also by having the hand positions arrays of both players already available from the human pose estimation performed prior to this processing, and simply by summing the modules of velocity in each frame.

The result achieved an (empirically observed) error of less than 250ms for most racket hits.

Other tests were performed with a cubic Savitzky-Golay filter but resulted in overfitting for high levels of noise.

The function *detect_racket_hits* implements all these approaches in a single function, taking the ball trajectory and player trajectory as input and returning a frame-reference array marking any estimated racket hit.

2.5.2 Distance travelled

This statistic was computed starting from the moving feet detection. Each time the feet were static, the centers of the positions of the players were saved into an array and the distance travelled was calculated as the length of the graph containing all the static centers in chronological order.

To compute the distance in meters, we computed the ratio pixel per meter by looking at the vertices of the field in the rectified image, by taking the distance between them to find the dimensions of the field in terms of pixels and by computing the proportion with the real dimensions of the field.

In this way we obtain a coefficient in pixels per meter. The next step is to simply divide the length of the graph containing all centers computed previously by the ratio.

2.5.3 Player's trajectory

This statistic is straightforward since we already have all the components to plot it; in fact, for each static center of each player, we plot a dot in the rectified image and if there is more than 1 static center, they are connected by a line. These last two statistics are computed in real time with the Human Pose Estimation.

2.5.4 Average ball speed between racket hits

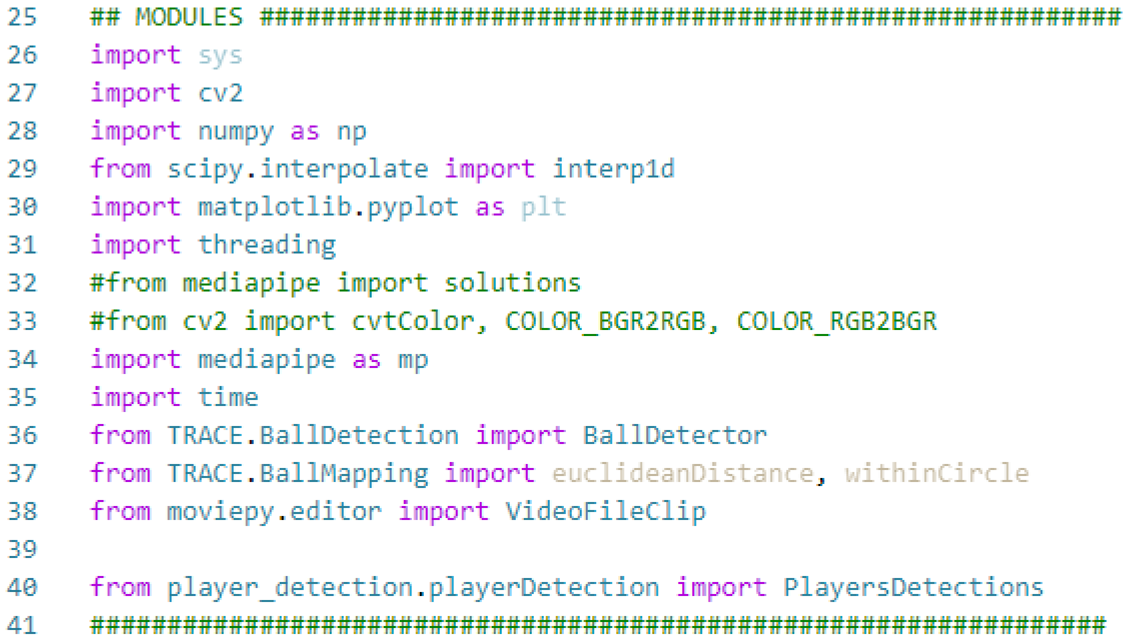
This statistic had the aim to compute the average ball speed resulting after a single racket hit up until the next one, by considering the distance in meters between the player that performs the hit and the player receiving at the time of racket response. To perform such computation, we’ve been required to consider the homographic transform of the two players’ positions at beginning and end of each 2-hit exchange, while computing the frame difference between the racket swings, and convert it in seconds with respect to the framerate.

The calculation performed is the basic physics formula to compute velocity, and the data obtained allows to show how much energy is put into the ball hit by the players.

3 Code implementation and description

In this chapter, the code utilized in the trajectory tracking of tennis players as well as in the computation of statistics based on the consecutive ball hits between the players will be presented and described accordingly.

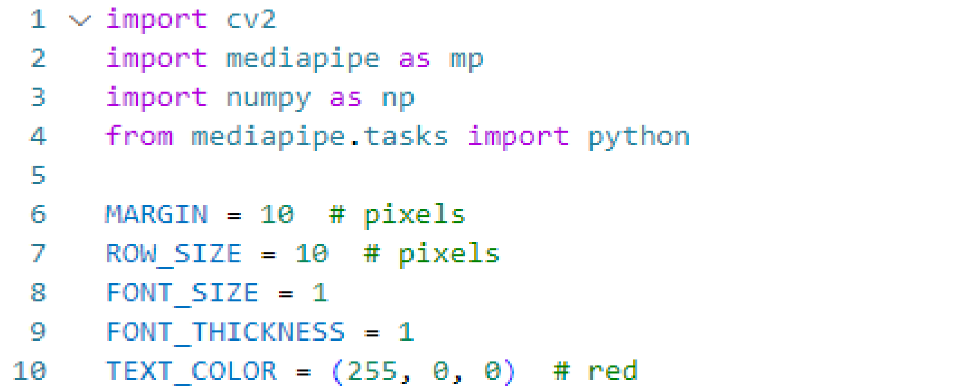
Note that this section aims to give a general understanding of how solutions were implemented. If any updates have been performed after the redaction of this section, or specific decisions of code implementations are to be checked, they are available and well-commented in the **GitHub /main.py** section of the code.

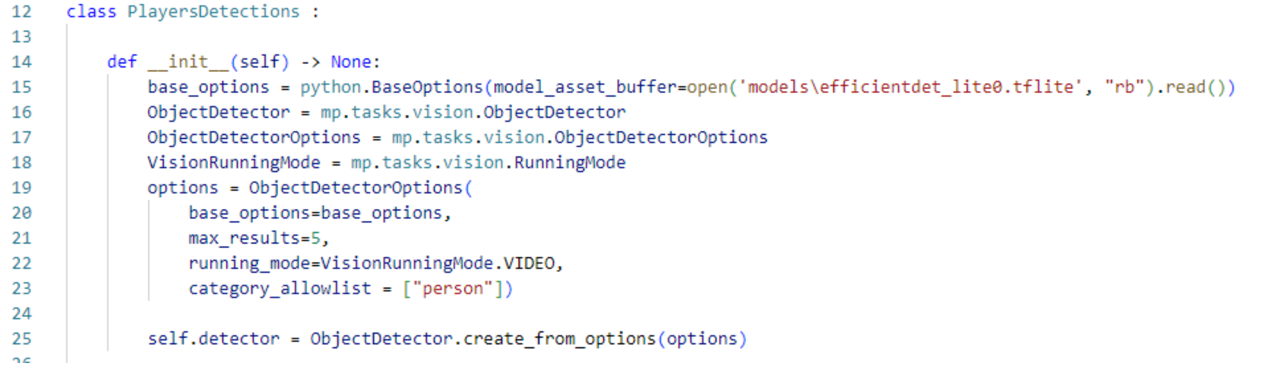


```
## MODULES #########################################################
import sys
import cv2
import numpy as np
from scipy.interpolate import interp1d
import matplotlib.pyplot as plt
import threading
#from mediapipe import solutions
#from cv2 import cvtColor, COLOR_BGR2RGB, COLOR_RGB2BGR
import mediapipe as mp
import time
from TRACE.BallDetection import BallDetector
from TRACE.BallMapping import euclideanDistance, withinCircle
from moviepy.editor import VideoFileClip

from player_detection.playerDetection import PlayersDetections
###################################################################
```

As a first step, the required libraries were imported along with our specific system path used for player detection and visualization.

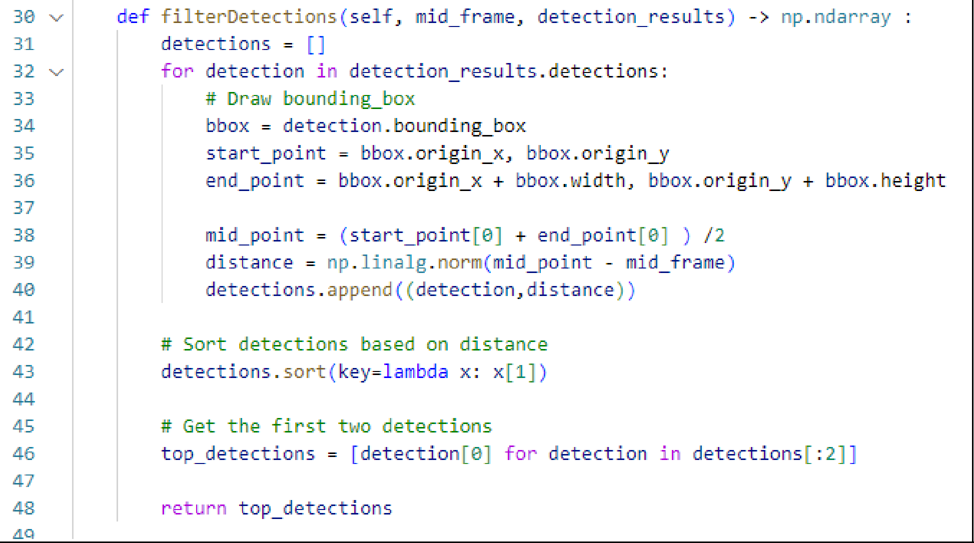
Table 3 Python Code from playerDetection.py

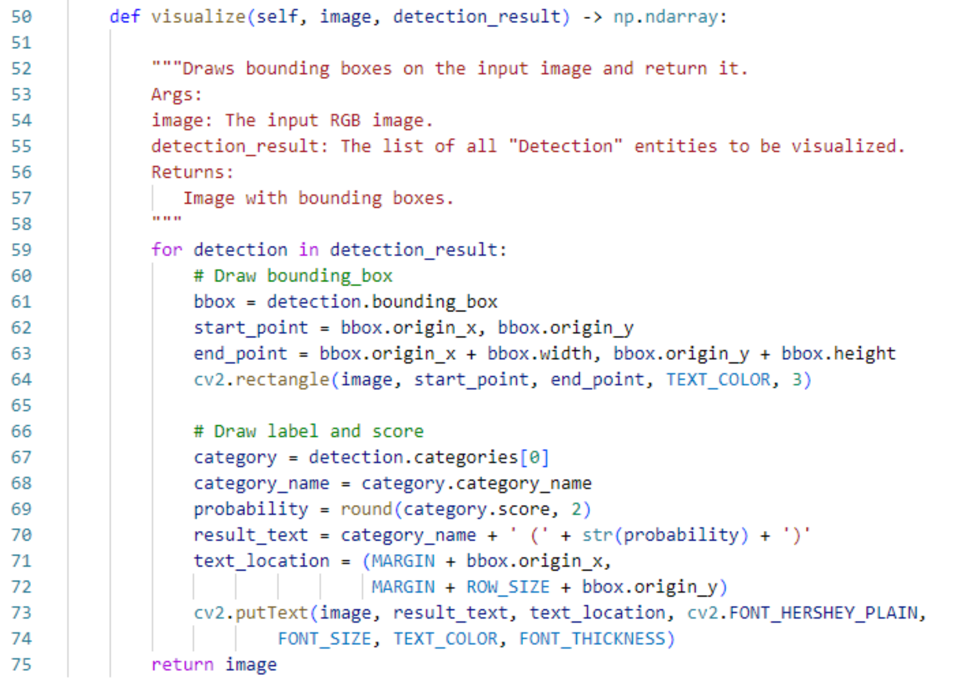

import cv2
import mediapipe as mp
import numpy as np
from mediapipe.tasks import python
 	
MARGIN = 10 # pixels
ROW_SIZE = 10 # pixels
FONT_SIZE = 1
FONT_THICKNESS = 1
TEXT_COLOR = (255, 0, 0) # red
	Libraries required for the implementation of the code.
	
	Constants used for drawing bounding boxes and text on the images


class PlayersDetections :
 	
    def __init__(self) -> None:
        base_options = python.BaseOptions(model_asset_buffer=open('models\efficientdet_lite0.tflite', "rb").read())
        ObjectDetector = mp.tasks.vision.ObjectDetector
        ObjectDetectorOptions = mp.tasks.vision.ObjectDetectorOptions
        VisionRunningMode = mp.tasks.vision.RunningMode
        options = ObjectDetectorOptions(
            base_options=base_options,
            max_results=5,
            running_mode=VisionRunningMode.VIDEO,
            category_allowlist = ["person"])
 	
        self.detector = ObjectDetector.create_from_options(options)	

Class-based implementation: **PlayerDetections** **“_init_”** Method – initialize the object detector with *EfficientDet Lite0* model (*efficientdet_lite0.tflite*) Configured to detect a maximum of 5 results, run in video mode and only allow detections of persons.	

def getDetector(self) :
return self.detector
	Return the initialized object detector


    def filterDetections(self, mid_frame, detection_results) -> np.ndarray :
        detections = []
        for detection in detection_results.detections:
            # Draw bounding_box
            bbox = detection.bounding_box
            start_point = bbox.origin_x, bbox.origin_y
            end_point = bbox.origin_x + bbox.width, bbox.origin_y + bbox.height
 	
            mid_point = (start_point[0] + end_point[0]) /2
            distance = np.linalg.norm(mid_point - mid_frame)
            detections.append((detection,distance))
 	
        # Sort detections based on distance
        detections.sort(key=lambda x: x[1])
 	
        # Get the first two detections
        top_detections = [detection[0] for detection in detections[:2]]
 	
        return top_detections
	Filter the detections based on their distance from the midpoint of the frame, then calculate the distance of each detected person from the center of the frame and sort the detections by distance. In the end, return the closest two detections.

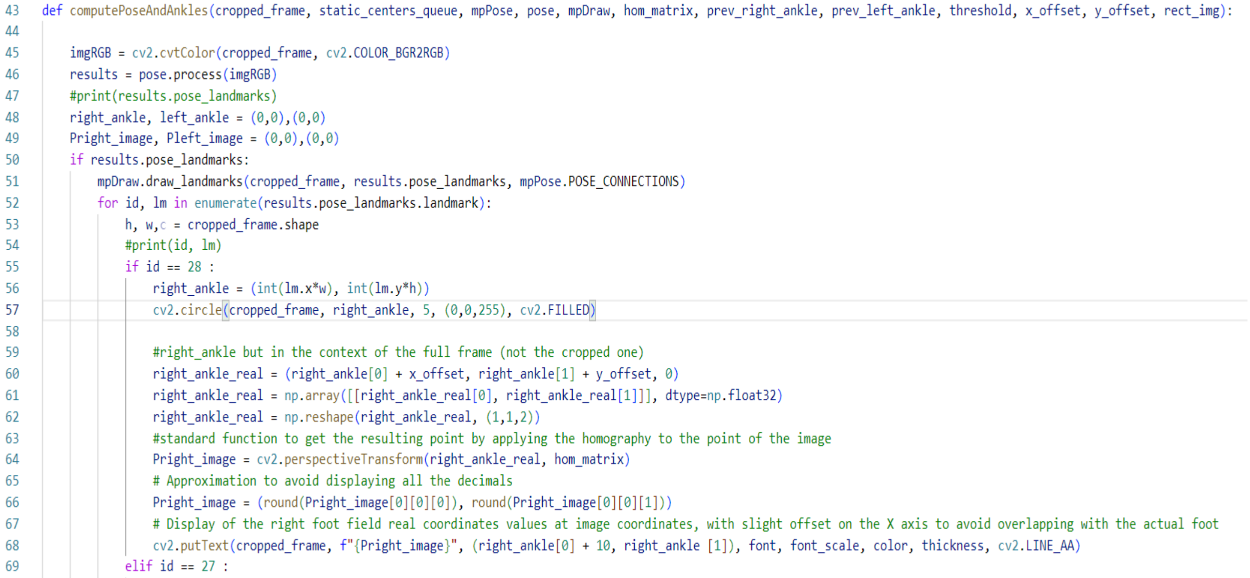


```python
    def visualize(self, image, detection_result) -> np.ndarray:

        """Draws bounding boxes on the input image and return it.
        Args:
        image: The input RGB image.
        detection_result: The list of all "Detection" entities to be visualized.
        Returns:
            Image with bounding boxes.
        """
        for detection in detection_result:
            # Draw bounding_box
            bbox = detection.bounding_box
            start_point = bbox.origin_x, bbox.origin_y
            end_point = bbox.origin_x + bbox.width, bbox.origin_y + bbox.height
            cv2.rectangle(image, start_point, end_point, TEXT_COLOR, 3)

            # Draw label and score
            category = detection.categories[0]
            category_name = category.category_name
            probability = round(category.score, 2)
            result_text = category_name + ' (' + str(probability) + ')'
            text_location = (MARGIN + bbox.origin_x,
                             MARGIN + ROW_SIZE + bbox.origin_y)
            cv2.putText(image, result_text, text_location, cv2.FONT_HERSHEY_PLAIN,
                        FONT_SIZE, TEXT_COLOR, FONT_THICKNESS)
        return image
```

Draw the bounding boxed around detected persons and add text annotations showing the category (person) and the detection confidence score.

As a result, class PlayersDetections is defined with the scope of detecting only the tennis players on the pitch (without the ball boys or the referee) using the pre-trained model *EfficientDet_Lite0* and leveraging the MediaPipe library for detection and OpenCV for visualization.

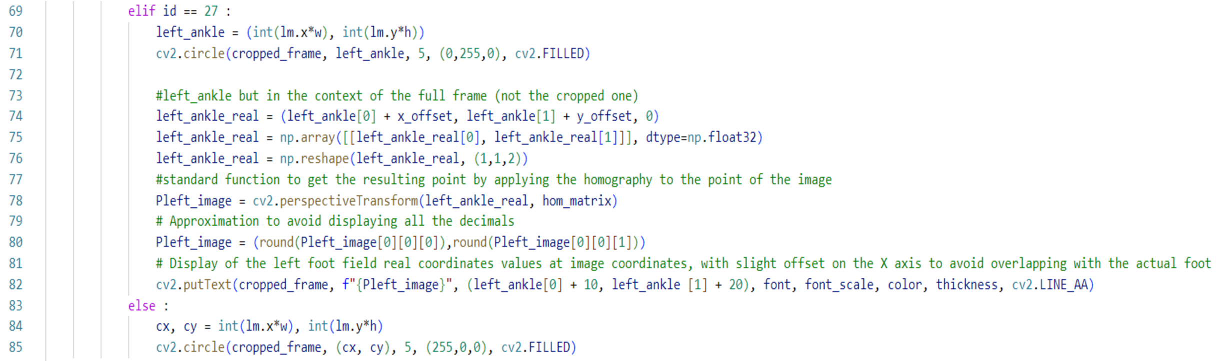
Now, getting back to *main.py* python code file, we have defined several functions to call in the main section of the code to improve the efficiency of the algorithm.



```python
def computePoseAndAnkles(cropped_frame, static_centers_queue, mpPose, pose, mpDraw, hom_matrix, prev_right_ankle, prev_left_ankle, threshold, x_offset, y_offset, rect_img):

    imgRGB = cv2.cvtColor(cropped_frame, cv2.COLOR_BGR2RGB)
    results = pose.process(imgRGB)
    #print(results.pose_landmarks)
    right_ankle, left_ankle = (0,0),(0,0)
    Pright_image, Pleft_image = (0,0),(0,0)
    if results.pose_landmarks:
        mpDraw.draw_landmarks(cropped_frame, results.pose_landmarks, mpPose.POSE_CONNECTIONS)
        for id, lm in enumerate(results.pose_landmarks.landmark):
            h, w,c = cropped_frame.shape
            #print(id, lm)
            if id == 28 :
                right_ankle = (int(lm.x*w), int(lm.y*h))
                cv2.circle(cropped_frame, right_ankle, 5, (0,0,255), cv2.FILLED)

                #right_ankle but in the context of the full frame (not the cropped one)
                right_ankle_real = (right_ankle[0] + x_offset, right_ankle[1] + y_offset, 0)
                right_ankle_real = np.array([[right_ankle_real[0], right_ankle_real[1]]], dtype=np.float32)
                right_ankle_real = np.reshape(right_ankle_real, (1,1,2))
                #standard function to get the resulting point by applying the homography to the point of the image
                Pright_image = cv2.perspectiveTransform(right_ankle_real, hom_matrix)
                # Approximation to avoid displaying all the decimals
                Pright_image = (round(Pright_image[0][0][0]), round(Pright_image[0][0][1]))
                # Display of the right foot field real coordinates values at image coordinates, with slight offset on the X axis to avoid overlapping with the actual foot
                cv2.putText(cropped_frame, f"{Pright_image}", (right_ankle[0] + 10, right_ankle [1]), font, font_scale, color, thickness, cv2.LINE_AA)
            elif id == 27 :
```

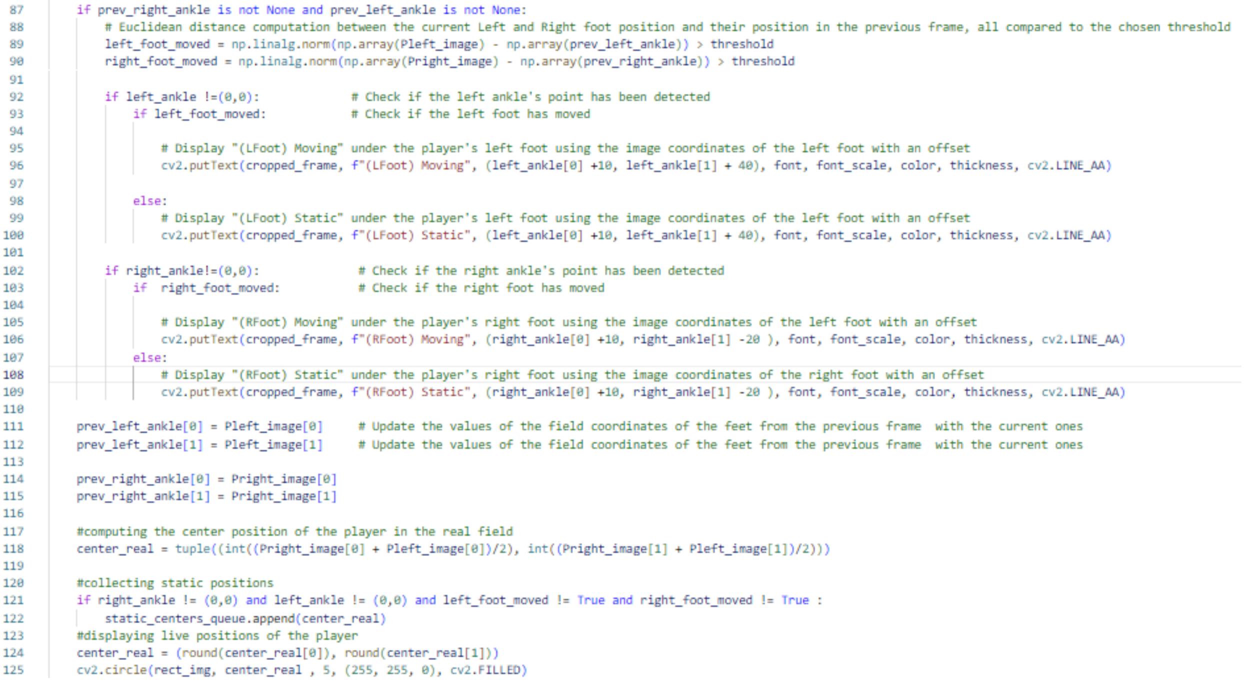
Function "computePoseAndAnkles" is defined as to detect player poses and their ankle positions (i.e. *human pose estimation*). It starts by processing a cropped frame (converting it from BGR to

RGB followed by detecting the pose landmarks through MediaPipe Pose function). If there is a positive detection of the pose landmarks, they are drawn on the cropped frame through the function *mpDraw,draw_landmarks*.



```
            elif id == 27 :
                left_ankle = (int(lm.x*w), int(lm.y*h))
                cv2.circle(cropped_frame, left_ankle, 5, (0,255,0), cv2.FILLED)

                #left_ankle but in the context of the full frame (not the cropped one)
                left_ankle_real = (left_ankle[0] + x_offset, left_ankle[1] + y_offset, 0)
                left_ankle_real = np.array([[left_ankle_real[0], left_ankle_real[1]]], dtype=np.float32)
                left_ankle_real = np.reshape(left_ankle_real, (1,1,2))
                #standard function to get the resulting point by applying the homography to the point of the image
                Pleft_image = cv2.perspectiveTransform(left_ankle_real, hom_matrix)
                # Approximation to avoid displaying all the decimals
                Pleft_image = (round(Pleft_image[0][0][0]),round(Pleft_image[0][0][1]))
                # Display of the left foot field real coordinates values at image coordinates, with slight offset on the X axis to avoid overlapping with the actual foot
                cv2.putText(cropped_frame, f"{Pleft_image}", (left_ankle[0] + 10, left_ankle [1] + 20), font, font_scale, color, thickness, cv2.LINE_AA)
            else :
                cx, cy = int(lm.x*w), int(lm.y*h)
                cv2.circle(cropped_frame, (cx, cy), 5, (255,0,0), cv2.FILLED)
```

Then, the positions of the right and left ankles (**id==27** and **id==28** in MediaPipe deep learning model's algorithm) are found and extracted to be later transformed, using the function *cv2.perspectiveTransform* with the homography matrix, H (later computed), to the real tennis field coordinates. Subsequently, the real coordinates of the tennis player's ankles are displayed.

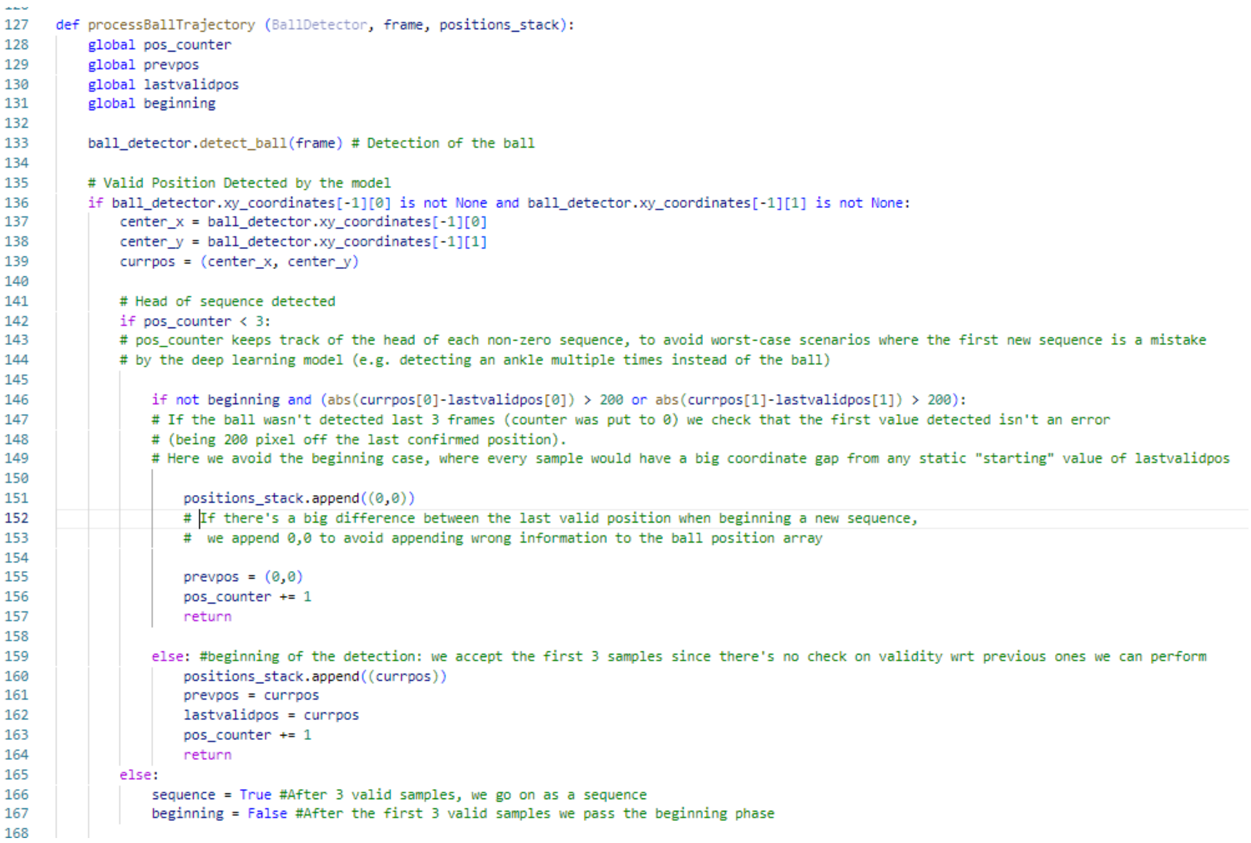


```
    if prev_right_ankle is not None and prev_left_ankle is not None:
        # Euclidean distance computation between the current Left and Right foot position and their position in the previous frame, all compared to the chosen threshold
        left_foot_moved = np.linalg.norm(np.array(Pleft_image) - np.array(prev_left_ankle)) > threshold
        right_foot_moved = np.linalg.norm(np.array(Pright_image) - np.array(prev_right_ankle)) > threshold

        if left_ankle !=(0,0):              # Check if the left ankle's point has been detected
            if left_foot_moved:             # Check if the left foot has moved

                # Display "(LFoot) Moving" under the player's left foot using the image coordinates of the left foot with an offset
                cv2.putText(cropped_frame, f"(LFoot) Moving", (left_ankle[0] +10, left_ankle[1] + 40), font, font_scale, color, thickness, cv2.LINE_AA)

            else:
                # Display "(LFoot) Static" under the player's left foot using the image coordinates of the left foot with an offset
                cv2.putText(cropped_frame, f"(LFoot) Static", (left_ankle[0] +10, left_ankle[1] + 40), font, font_scale, color, thickness, cv2.LINE_AA)

        if right_ankle!=(0,0):               # Check if the right ankle's point has been detected
            if  right_foot_moved:            # Check if the right foot has moved

                # Display "(RFoot) Moving" under the player's right foot using the image coordinates of the left foot with an offset
                cv2.putText(cropped_frame, f"(RFoot) Moving", (right_ankle[0] +10, right_ankle[1] -20 ), font, font_scale, color, thickness, cv2.LINE_AA)
            else:
                # Display "(RFoot) Static" under the player's right foot using the image coordinates of the right foot with an offset
                cv2.putText(cropped_frame, f"(RFoot) Static", (right_ankle[0] +10, right_ankle[1] -20 ), font, font_scale, color, thickness, cv2.LINE_AA)

    prev_left_ankle[0] = Pleft_image[0]      # Update the values of the field coordinates of the feet from the previous frame  with the current ones
    prev_left_ankle[1] = Pleft_image[1]      # Update the values of the field coordinates of the feet from the previous frame  with the current ones

    prev_right_ankle[0] = Pright_image[0]
    prev_right_ankle[1] = Pright_image[1]

    #computing the center position of the player in the real field
    center_real = tuple((int((Pright_image[0] + Pleft_image[0])/2), int((Pright_image[1] + Pleft_image[1])/2)))

    #collecting static positions
    if right_ankle != (0,0) and left_ankle != (0,0) and left_foot_moved != True and right_foot_moved != True :
        static_centers_queue.append(center_real)
    #displaying live positions of the player
    center_real = (round(center_real[0]), round(center_real[1]))
    cv2.circle(rect_img, center_real , 5, (255, 255, 0), cv2.FILLED)
```

Based on the last lines of code of the function, the algorithm continues to work through the following steps:

- Determine the dynamics of the feet (i.e. whether one of the feet has moved or not with respect to the previous frame) through the computation of the Euclidean distance ($np.linalg.norm = \sqrt{(x1 - x2)^2 + (y1 - y2)^2}$);
- Display the message "Moving" under the moving foot or "Static" otherwise;
- Update the previous frame positions with the current ones;

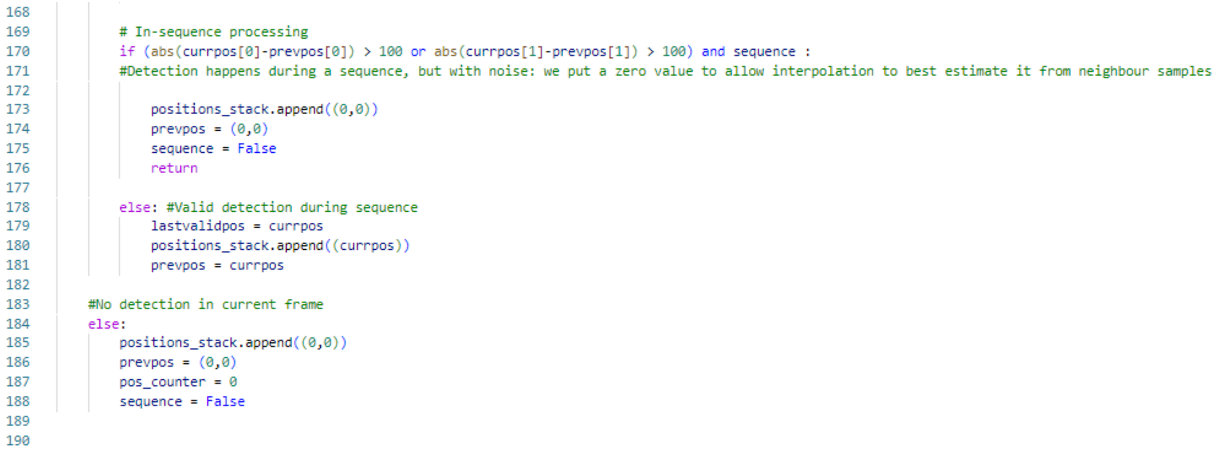
- Compute the center position of the player in the real tennis field;
- Compute the center position of the player in the field (i.e. coordinate between the two moving feet of the player) and display the live position of the player;
- Finally, collect the static position of the feet of the player (KEY step in the computation of the tennis players' trajectory.



```python
def processBallTrajectory (BallDetector, frame, positions_stack):
    global pos_counter
    global prevpos
    global lastvalidpos
    global beginning

    ball_detector.detect_ball(frame) # Detection of the ball

    # Valid Position Detected by the model
    if ball_detector.xy_coordinates[-1][0] is not None and ball_detector.xy_coordinates[-1][1] is not None:
        center_x = ball_detector.xy_coordinates[-1][0]
        center_y = ball_detector.xy_coordinates[-1][1]
        currpos = (center_x, center_y)

        # Head of sequence detected
        if pos_counter < 3:
        # pos_counter keeps track of the head of each non-zero sequence, to avoid worst-case scenarios where the first new sequence is a mistake
        # by the deep learning model (e.g. detecting an ankle multiple times instead of the ball)

            if not beginning and (abs(currpos[0]-lastvalidpos[0]) > 200 or abs(currpos[1]-lastvalidpos[1]) > 200):
            # If the ball wasn't detected last 3 frames (counter was put to 0) we check that the first value detected isn't an error
            # (being 200 pixel off the last confirmed position).
            # Here we avoid the beginning case, where every sample would have a big coordinate gap from any static "starting" value of lastvalidpos

                positions_stack.append((0,0))
                # If there's a big difference between the last valid position when beginning a new sequence,
                #  we append 0,0 to avoid appending wrong information to the ball position array

                prevpos = (0,0)
                pos_counter += 1
                return

            else: #beginning of the detection: we accept the first 3 samples since there's no check on validity wrt previous ones we can perform
                positions_stack.append((currpos))
                prevpos = currpos
                lastvalidpos = currpos
                pos_counter += 1
                return
        else:
            sequence = True #After 3 valid samples, we go on as a sequence
            beginning = False #After the first 3 valid samples we pass the beginning phase

```

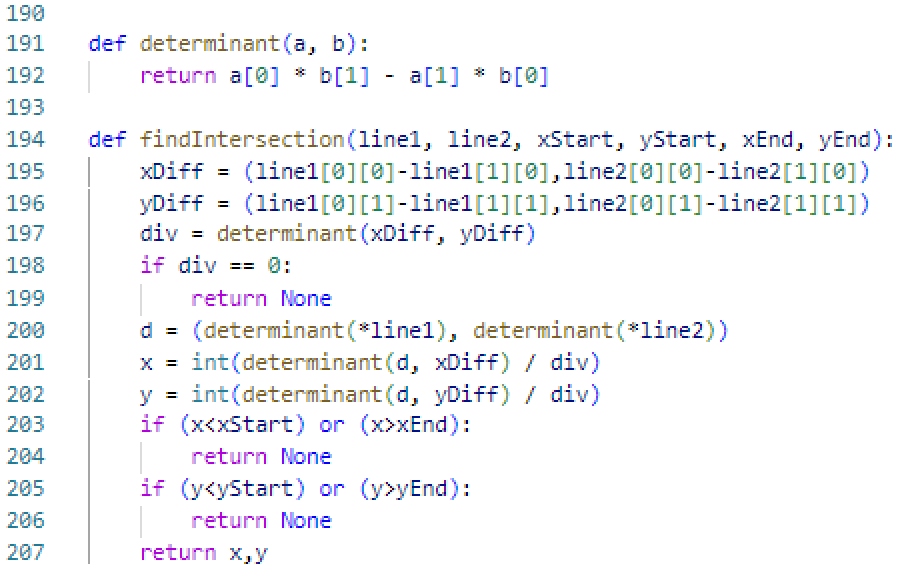
Arriving at ball detection algorithm, "processBallTrajectory function is defined to process the trajectory of the ball in the frame by detecting its position using the previously mentioned for ball detection called TRACE.

The algorithm works by handling valid ball detections by comparing the current ball position with the last valid position. However, a sequence counter is used in order to filter out noisy detections (ankle detection, ball detection with an offset > 200px with respect to the previously valid position found).



```
        # In-sequence processing
        if (abs(currpos[0]-prevpos[0]) > 100 or abs(currpos[1]-prevpos[1]) > 100) and sequence :
        #Detection happens during a sequence, but with noise: we put a zero value to allow interpolation to best estimate it from neighbour samples

            positions_stack.append((0,0))
            prevpos = (0,0)
            sequence = False
            return

        else: #Valid detection during sequence
            lastvalidpos = currpos
            positions_stack.append((currpos))
            prevpos = currpos

    #No detection in current frame
    else:
        positions_stack.append((0,0))
        prevpos = (0,0)
        pos_counter = 0
        sequence = False
```

Subsequently, the valid ball positions are appended to a position stack. As a safety measure, we handle frames with no detections by appending a placeholder value.



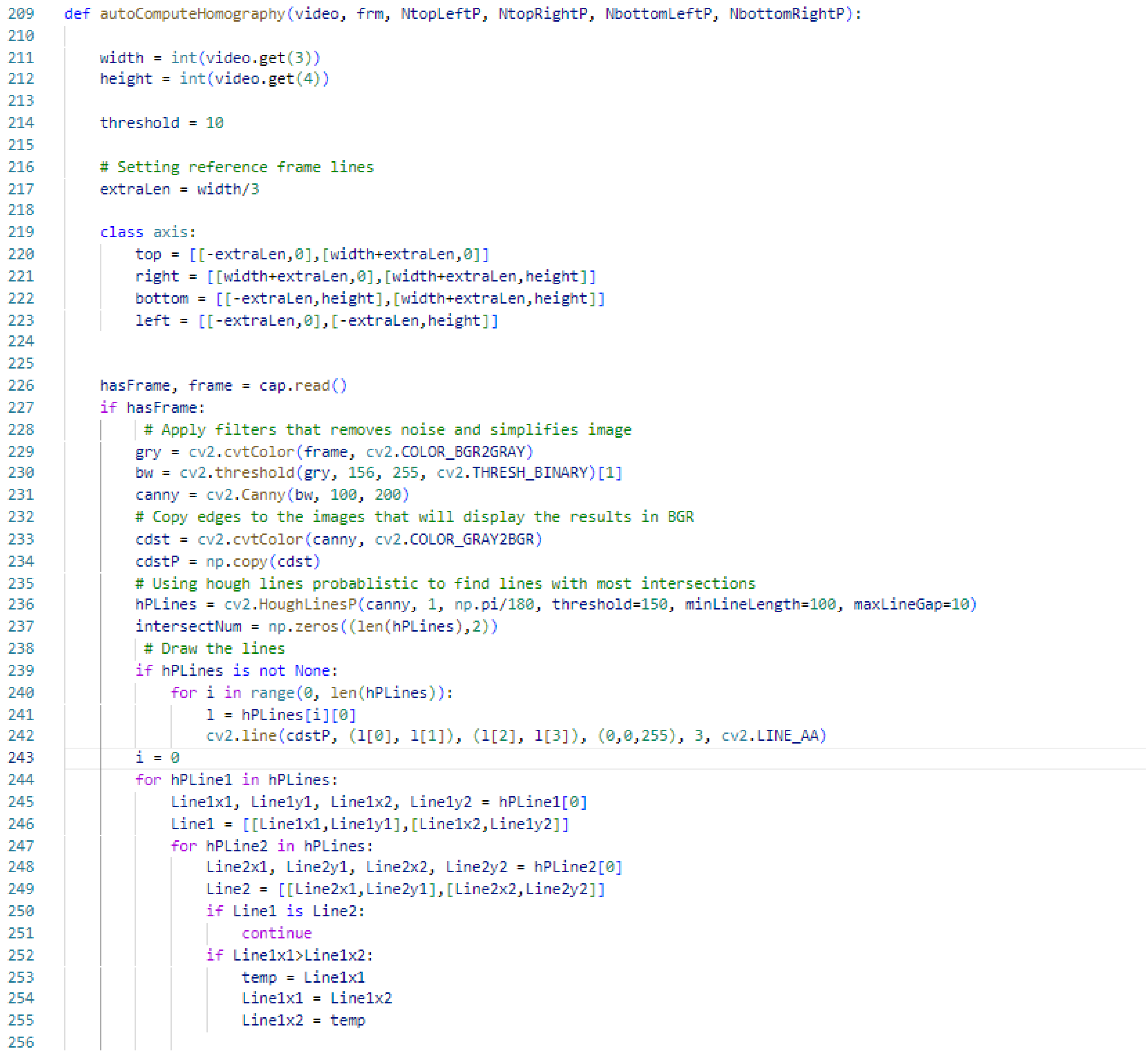
```
def determinant(a, b):
    return a[0] * b[1] - a[1] * b[0]

def findIntersection(line1, line2, xStart, yStart, xEnd, yEnd):
    xDiff = (line1[0][0]-line1[1][0],line2[0][0]-line2[1][0])
    yDiff = (line1[0][1]-line1[1][1],line2[0][1]-line2[1][1])
    div = determinant(xDiff, yDiff)
    if div == 0:
        return None
    d = (determinant(*line1), determinant(*line2))
    x = int(determinant(d, xDiff) / div)
    y = int(determinant(d, yDiff) / div)
    if (x<xStart) or (x>xEnd):
        return None
    if (y<yStart) or (y>yEnd):
        return None
    return x,y
```

Arriving at *determinant* and *findIntersection* functions, these two are support functions defined to help in the computation of the later defined function having the role to compute the Homography matrix as previously mentioned.

As such, *determinant* function, as the name suggests, calculates the determinant of two vectors.

Function *findIntersection*, suggestively again, finds the intersection point of two lines within specified bounds in the following way:

- Calculates the difference between the X and Y coordinates of the lines and calls function *determinant* to check whether the lines are parallel or not;
- x and y become the coordinates of the intersection point which is further checked if it lies within the specified bounds.



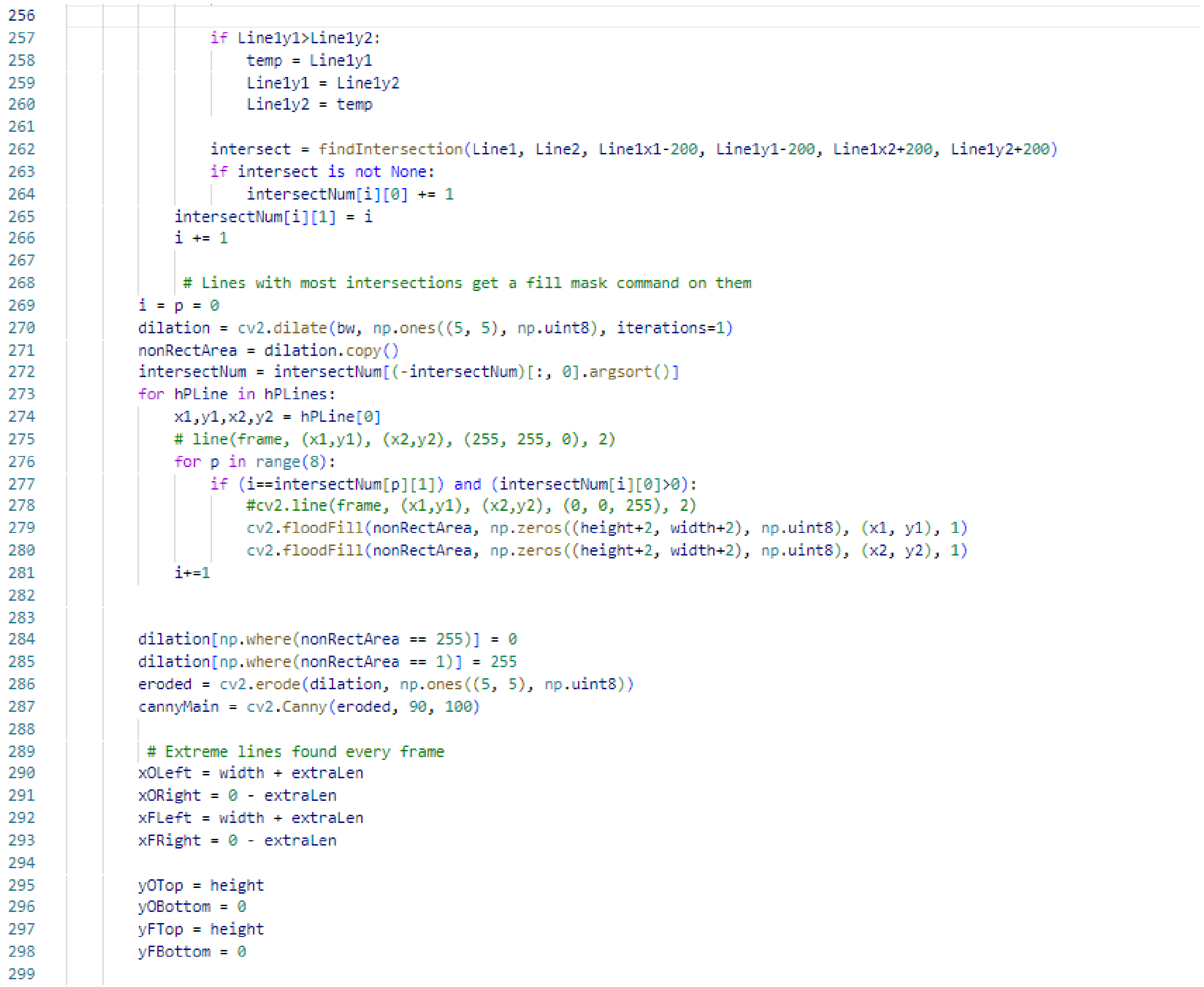
```
def autoComputeHomography(video, frm, NtopLeftP, NtopRightP, NbottomLeftP, NbottomRightP):

    width = int(video.get(3))
    height = int(video.get(4))

    threshold = 10

    # Setting reference frame lines
    extraLen = width/3

    class axis:
        top = [[-extraLen,0],[width+extraLen,0]]
        right = [[width+extraLen,0],[width+extraLen,height]]
        bottom = [[-extraLen,height],[width+extraLen,height]]
        left = [[-extraLen,0],[-extraLen,height]]


    hasFrame, frame = cap.read()
    if hasFrame:
         # Apply filters that removes noise and simplifies image
        gry = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        bw = cv2.threshold(gry, 156, 255, cv2.THRESH_BINARY)[1]
        canny = cv2.Canny(bw, 100, 200)
        # Copy edges to the images that will display the results in BGR
        cdst = cv2.cvtColor(canny, cv2.COLOR_GRAY2BGR)
        cdstP = np.copy(cdst)
        # Using hough lines probablistic to find lines with most intersections
        hPLines = cv2.HoughLinesP(canny, 1, np.pi/180, threshold=150, minLineLength=100, maxLineGap=10)
        intersectNum = np.zeros((len(hPLines),2))
         # Draw the lines
        if hPLines is not None:
            for i in range(0, len(hPLines)):
                l = hPLines[i][0]
                cv2.line(cdstP, (l[0], l[1]), (l[2], l[3]), (0,0,255), 3, cv2.LINE_AA)
        i = 0
        for hPLine1 in hPLines:
            Line1x1, Line1y1, Line1x2, Line1y2 = hPLine1[0]
            Line1 = [[Line1x1,Line1y1],[Line1x2,Line1y2]]
            for hPLine2 in hPLines:
                Line2x1, Line2y1, Line2x2, Line2y2 = hPLine2[0]
                Line2 = [[Line2x1,Line2y1],[Line2x2,Line2y2]]
                if Line1 is Line2:
                    continue
                if Line1x1>Line1x2:
                    temp = Line1x1
                    Line1x1 = Line1x2
                    Line1x2 = temp

```

The function *autoComputeHomography* is the function responsible for computing the homography matrix for mapping the court by processing a frame and detecting the court lines.

It does so by:

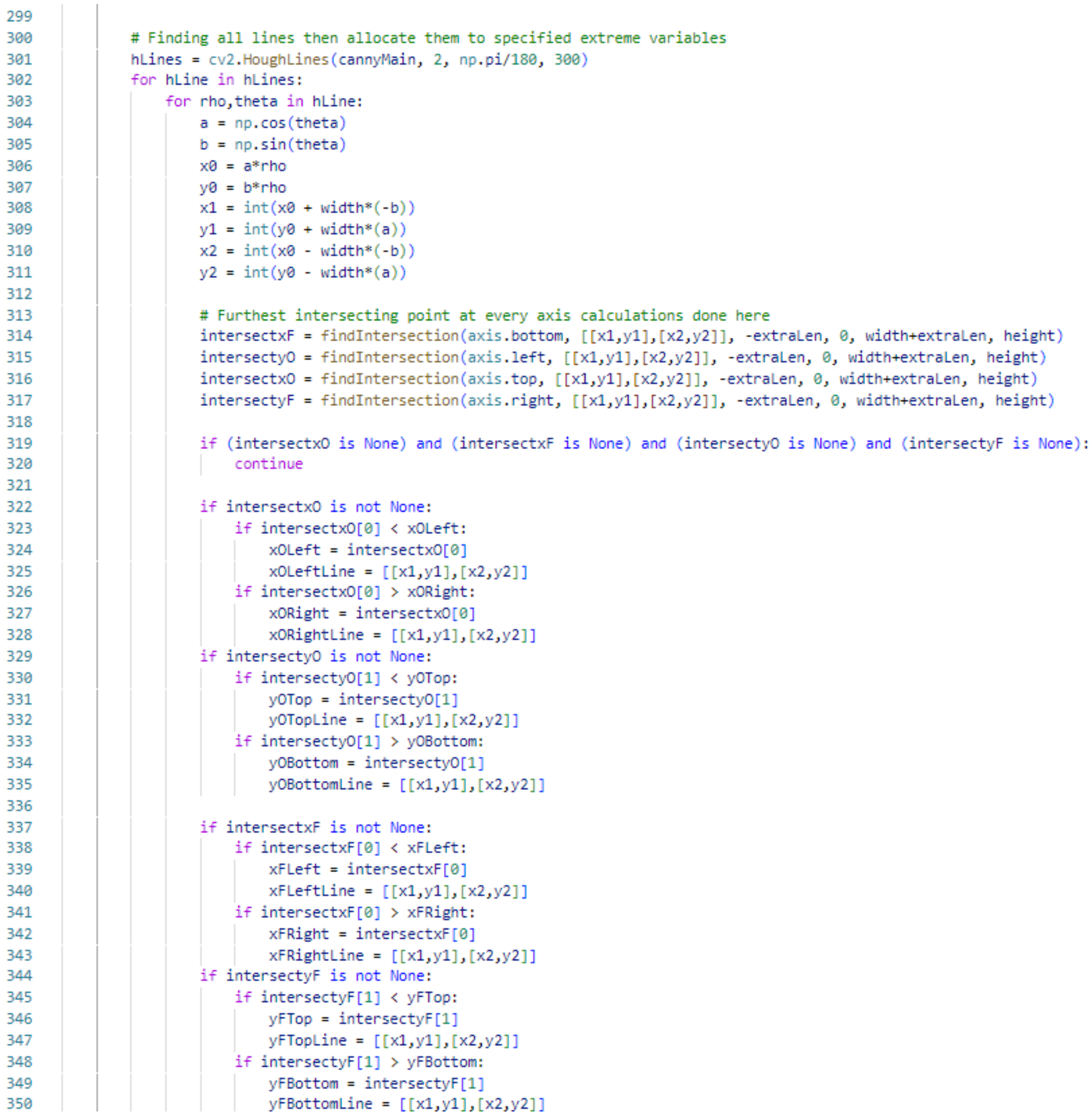
- setting a threshold parameter and retrieving the dimensions of the frame of the video;
- setting reference lines in class *axis* with a positive offset (*extraLen*) from the retrieved dimensions;
- reading the frame and pre-processing it through a conversion to grayscale, applying a binary thresholding (all pixels with a value greater than 156 are set to 255/white and the rest to 0/black) and afterwards, applying Canny edge detection; the aforementioned methods are then followed by converting the obtained edge image back to BGR color space;
- detecting lines with the highest number of intersections through Hough Transform (*cv2.HoughLinesP*) and then drawing them;

- from line 243 ->266: finding intersections of lines using the previously defined function *findIntersection.*



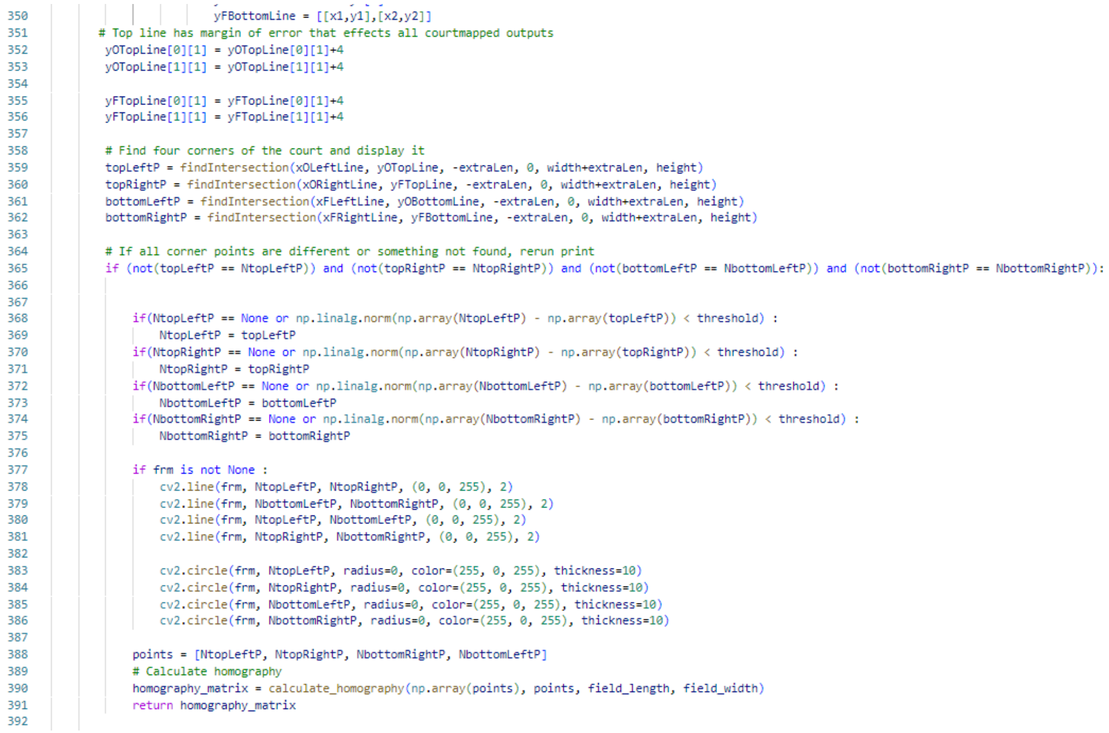
```
256
257                 if Line1y1>Line1y2:
258                     temp = Line1y1
259                     Line1y1 = Line1y2
260                     Line1y2 = temp
261
262                 intersect = findIntersection(Line1, Line2, Line1x1-200, Line1y1-200, Line1x2+200, Line1y2+200)
263                 if intersect is not None:
264                     intersectNum[i][0] += 1
265             intersectNum[i][1] = i
266             i += 1
267
268              # Lines with most intersections get a fill mask command on them
269         i = p = 0
270         dilation = cv2.dilate(bw, np.ones((5, 5), np.uint8), iterations=1)
271         nonRectArea = dilation.copy()
272         intersectNum = intersectNum[(-intersectNum)[:, 0].argsort()]
273         for hPLine in hPLines:
274             x1,y1,x2,y2 = hPLine[0]
275             # line(frame, (x1,y1), (x2,y2), (255, 255, 0), 2)
276             for p in range(8):
277                 if (i==intersectNum[p][1]) and (intersectNum[i][0]>0):
278                     #cv2.line(frame, (x1,y1), (x2,y2), (0, 0, 255), 2)
279                     cv2.floodFill(nonRectArea, np.zeros((height+2, width+2), np.uint8), (x1, y1), 1)
280                     cv2.floodFill(nonRectArea, np.zeros((height+2, width+2), np.uint8), (x2, y2), 1)
281             i+=1
282
283
284         dilation[np.where(nonRectArea == 255)] = 0
285         dilation[np.where(nonRectArea == 1)] = 255
286         eroded = cv2.erode(dilation, np.ones((5, 5), np.uint8))
287         cannyMain = cv2.Canny(eroded, 90, 100)
288
289          # Extreme lines found every frame
290         xOLeft = width + extraLen
291         xORight = 0 - extraLen
292         xFLeft = width + extraLen
293         xFRight = 0 - extraLen
294
295         yOTop = height
296         yOBottom = 0
297         yFTop = height
298         yFBottom = 0
299
```

- fill masking the lines with most intersections;



```python

        # Finding all lines then allocate them to specified extreme variables
        hLines = cv2.HoughLines(cannyMain, 2, np.pi/180, 300)
        for hLine in hLines:
            for rho,theta in hLine:
                a = np.cos(theta)
                b = np.sin(theta)
                x0 = a*rho
                y0 = b*rho
                x1 = int(x0 + width*(-b))
                y1 = int(y0 + width*(a))
                x2 = int(x0 - width*(-b))
                y2 = int(y0 - width*(a))

                # Furthest intersecting point at every axis calculations done here
                intersectxF = findIntersection(axis.bottom, [[x1,y1],[x2,y2]], -extraLen, 0, width+extraLen, height)
                intersectyO = findIntersection(axis.left, [[x1,y1],[x2,y2]], -extraLen, 0, width+extraLen, height)
                intersectxO = findIntersection(axis.top, [[x1,y1],[x2,y2]], -extraLen, 0, width+extraLen, height)
                intersectyF = findIntersection(axis.right, [[x1,y1],[x2,y2]], -extraLen, 0, width+extraLen, height)

                if (intersectxO is None) and (intersectxF is None) and (intersectyO is None) and (intersectyF is None):
                    continue

                if intersectxO is not None:
                    if intersectxO[0] < xOLeft:
                        xOLeft = intersectxO[0]
                        xOLeftLine = [[x1,y1],[x2,y2]]
                    if intersectxO[0] > xORight:
                        xORight = intersectxO[0]
                        xORightLine = [[x1,y1],[x2,y2]]
                if intersectyO is not None:
                    if intersectyO[1] < yOTop:
                        yOTop = intersectyO[1]
                        yOTopLine = [[x1,y1],[x2,y2]]
                    if intersectyO[1] > yOBottom:
                        yOBottom = intersectyO[1]
                        yOBottomLine = [[x1,y1],[x2,y2]]

                if intersectxF is not None:
                    if intersectxF[0] < xFLeft:
                        xFLeft = intersectxF[0]
                        xFLeftLine = [[x1,y1],[x2,y2]]
                    if intersectxF[0] > xFRight:
                        xFRight = intersectxF[0]
                        xFRightLine = [[x1,y1],[x2,y2]]
                if intersectyF is not None:
                    if intersectyF[1] < yFTop:
                        yFTop = intersectyF[1]
                        yFTopLine = [[x1,y1],[x2,y2]]
                    if intersectyF[1] > yFBottom:
                        yFBottom = intersectyF[1]
                        yFBottomLine = [[x1,y1],[x2,y2]]
```

- finding extreme lines and intersections with the same process as before, now taking into consideration the previously computed reference lines;



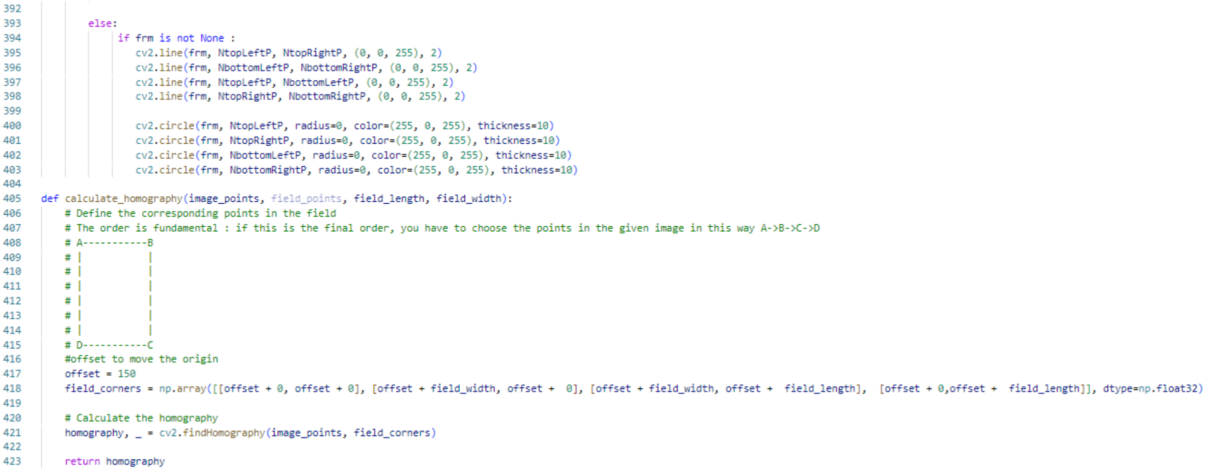
```
                        yFBottomLine = [[x1,y1],[x2,y2]]
       # Top line has margin of error that effects all courtmapped outputs
        yOTopLine[0][1] = yOTopLine[0][1]+4
        yOTopLine[1][1] = yOTopLine[1][1]+4

        yFTopLine[0][1] = yFTopLine[0][1]+4
        yFTopLine[1][1] = yFTopLine[1][1]+4

        # Find four corners of the court and display it
        topLeftP = findIntersection(xOLeftLine, yOTopLine, -extraLen, 0, width+extraLen, height)
        topRightP = findIntersection(xORightLine, yFTopLine, -extraLen, 0, width+extraLen, height)
        bottomLeftP = findIntersection(xFLeftLine, yOBottomLine, -extraLen, 0, width+extraLen, height)
        bottomRightP = findIntersection(xFRightLine, yFBottomLine, -extraLen, 0, width+extraLen, height)

        # If all corner points are different or something not found, rerun print
        if (not(topLeftP == NtopLeftP)) and (not(topRightP == NtopRightP)) and (not(bottomLeftP == NbottomLeftP)) and (not(bottomRightP == NbottomRightP)):


            if(NtopLeftP == None or np.linalg.norm(np.array(NtopLeftP) - np.array(topLeftP)) < threshold) :
                NtopLeftP = topLeftP
            if(NtopRightP == None or np.linalg.norm(np.array(NtopRightP) - np.array(topRightP)) < threshold) :
                NtopRightP = topRightP
            if(NbottomLeftP == None or np.linalg.norm(np.array(NbottomLeftP) - np.array(bottomLeftP)) < threshold) :
                NbottomLeftP = bottomLeftP
            if(NbottomRightP == None or np.linalg.norm(np.array(NbottomRightP) - np.array(bottomRightP)) < threshold) :
                NbottomRightP = bottomRightP

            if frm is not None :
                cv2.line(frm, NtopLeftP, NtopRightP, (0, 0, 255), 2)
                cv2.line(frm, NbottomLeftP, NbottomRightP, (0, 0, 255), 2)
                cv2.line(frm, NtopLeftP, NbottomLeftP, (0, 0, 255), 2)
                cv2.line(frm, NtopRightP, NbottomRightP, (0, 0, 255), 2)

                cv2.circle(frm, NtopLeftP, radius=0, color=(255, 0, 255), thickness=10)
                cv2.circle(frm, NtopRightP, radius=0, color=(255, 0, 255), thickness=10)
                cv2.circle(frm, NbottomLeftP, radius=0, color=(255, 0, 255), thickness=10)
                cv2.circle(frm, NbottomRightP, radius=0, color=(255, 0, 255), thickness=10)

            points = [NtopLeftP, NtopRightP, NbottomRightP, NbottomLeftP]
            # Calculate homography
            homography_matrix = calculate_homography(np.array(points), points, field_length, field_width)
            return homography_matrix

```

All the pre-processing and intersections determination has been used to be able to determine the tennis court’s corners.

Subsequently, the top, bottom, left and right extreme lines are identified, and the four corners of the court are calculated in *topLeftP, topRightP, bottomLeftP, bottomRightP*.

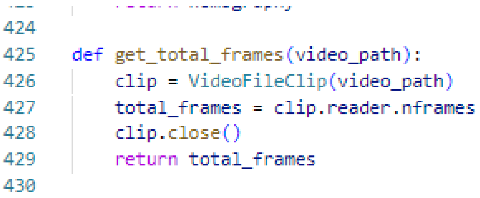
To be noted that the top line has a margin of error that affects all the court mapped outputs.

Additionally, the function *calculate_homography (*shown below) is used to finish the algorithm of the automated computation of the homography matrix.



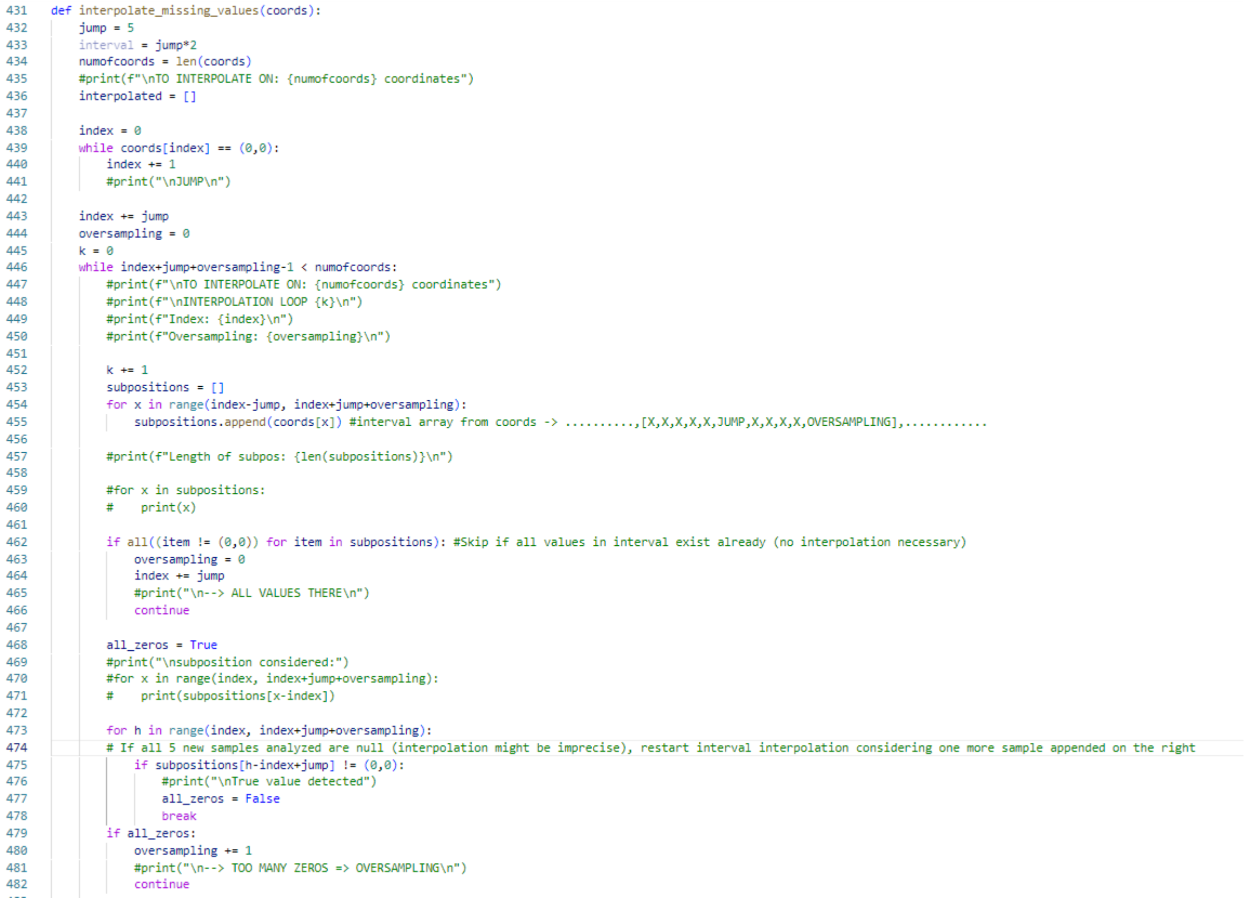
```

        else:
            if frm is not None :
                cv2.line(frm, NtopLeftP, NtopRightP, (0, 0, 255), 2)
                cv2.line(frm, NbottomLeftP, NbottomRightP, (0, 0, 255), 2)
                cv2.line(frm, NtopLeftP, NbottomLeftP, (0, 0, 255), 2)
                cv2.line(frm, NtopRightP, NbottomRightP, (0, 0, 255), 2)

                cv2.circle(frm, NtopLeftP, radius=0, color=(255, 0, 255), thickness=10)
                cv2.circle(frm, NtopRightP, radius=0, color=(255, 0, 255), thickness=10)
                cv2.circle(frm, NbottomLeftP, radius=0, color=(255, 0, 255), thickness=10)
                cv2.circle(frm, NbottomRightP, radius=0, color=(255, 0, 255), thickness=10)

def calculate_homography(image_points, field_points, field_length, field_width):
    # Define the corresponding points in the field
    # The order is fundamental : if this is the final order, you have to choose the points in the given image in this way A->B->C->D
    # A-----------B
    # |           |
    # |           |
    # |           |
    # |           |
    # |           |
    # |           |
    # D-----------C
    #offset to move the origin
    offset = 150
    field_corners = np.array([[offset + 0, offset + 0], [offset + field_width, offset +  0], [offset + field_width, offset +  field_length],  [offset + 0,offset +  field_length]], dtype=np.float32)

    # Calculate the homography
    homography, _ = cv2.findHomography(image_points, field_corners)

    return homography
```

Using four corners of the court, the homography matrix is computed while the identified points are matched with the corresponding real tennis field corners.



```python

def get_total_frames(video_path):
    clip = VideoFileClip(video_path)
    total_frames = clip.reader.nframes
    clip.close()
    return total_frames

```

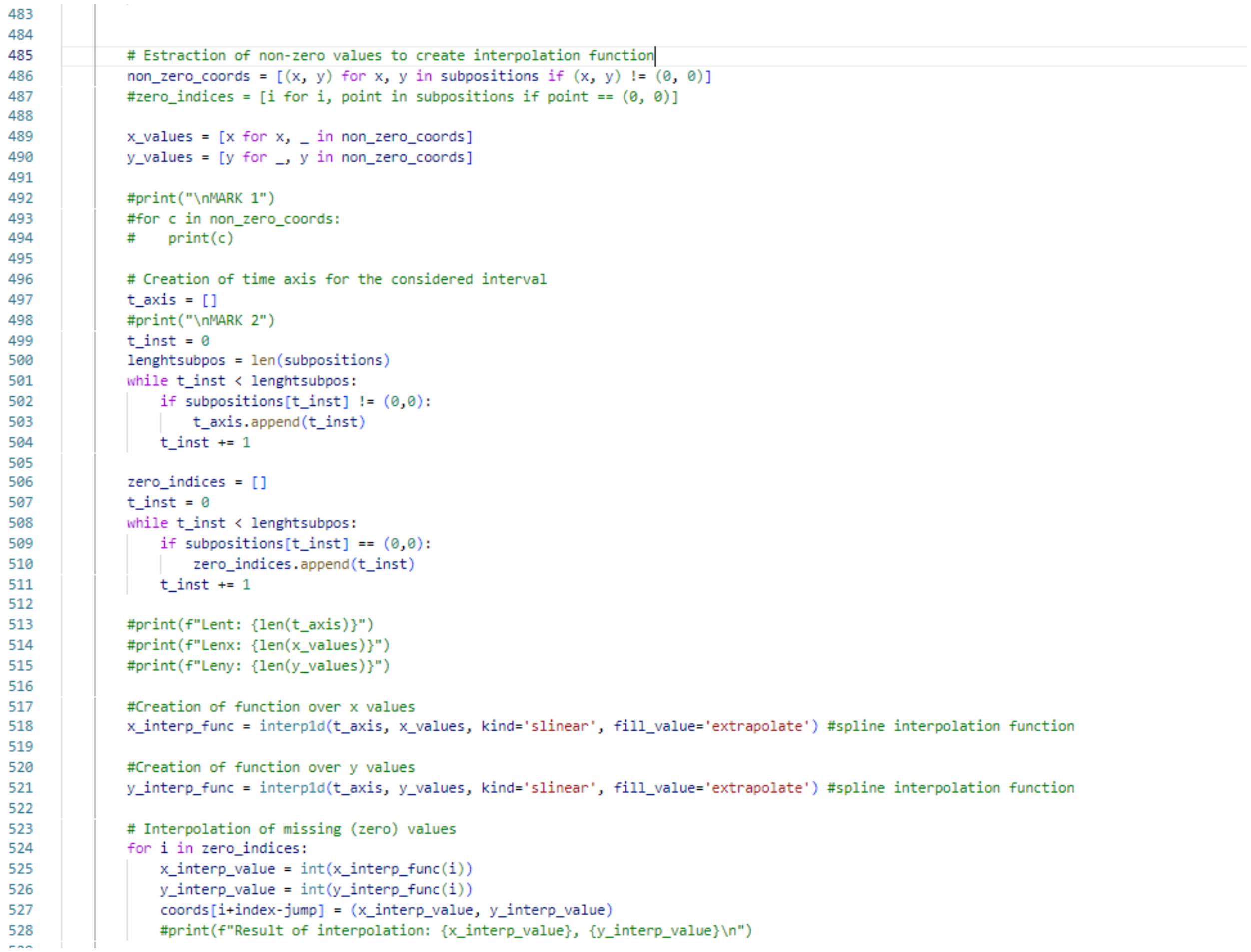
Moving forward, a function that calculates the total number of frames of the video is defined which will later be called when loading the input video to be analyzed.



```python
def interpolate_missing_values(coords):
    jump = 5
    interval = jump*2
    numofcoords = len(coords)
    #print(f"\nTO INTERPOLATE ON: {numofcoords} coordinates")
    interpolated = []

    index = 0
    while coords[index] == (0,0):
        index += 1
        #print("\nJUMP\n")

    index += jump
    oversampling = 0
    k = 0
    while index+jump+oversampling-1 < numofcoords:
        #print(f"\nTO INTERPOLATE ON: {numofcoords} coordinates")
        #print(f"\nINTERPOLATION LOOP {k}\n")
        #print(f"Index: {index}\n")
        #print(f"Oversampling: {oversampling}\n")

        k += 1
        subpositions = []
        for x in range(index-jump, index+jump+oversampling):
            subpositions.append(coords[x]) #interval array from coords -> .........,[X,X,X,X,X,JUMP,X,X,X,X,OVERSAMPLING],...........

        #print(f"Length of subpos: {len(subpositions)}\n")

        #for x in subpositions:
        #    print(x)

        if all((item != (0,0)) for item in subpositions): #Skip if all values in interval exist already (no interpolation necessary)
            oversampling = 0
            index += jump
            #print("\n--> ALL VALUES THERE\n")
            continue

        all_zeros = True
        #print("\nsubposition considered:")
        #for x in range(index, index+jump+oversampling):
        #    print(subpositions[x-index])

        for h in range(index, index+jump+oversampling):
        # If all 5 new samples analyzed are null (interpolation might be imprecise), restart interval interpolation considering one more sample appended on the right
            if subpositions[h-index+jump] != (0,0):
                #print("\nTrue value detected")
                all_zeros = False
                break
        if all_zeros:
            oversampling += 1
            #print("\n--> TOO MANY ZEROS => OVERSAMPLING\n")
            continue
```

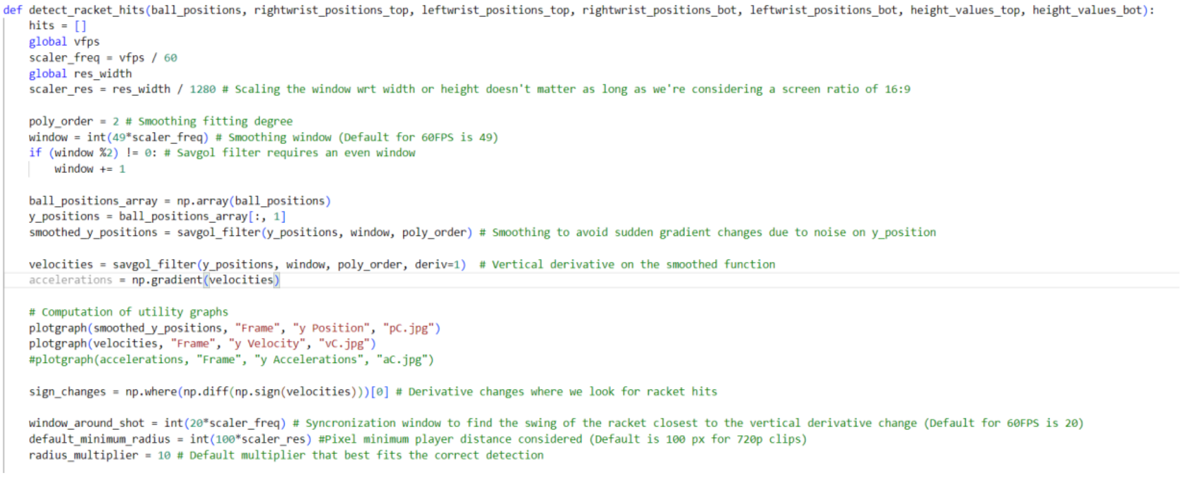
An important addition to the code used is the definition of the function *interpolate_missing_values* which fills in missing coordinate values represented as (0,0) in the coordinates list.

Several variables such as jump, interval, index, oversampling and k are initialized. Furthermore, the algorithm finds the initial non-zero coordinates and increases the index value with one for each (0,0) coordinate found up until finding the initial non-zero coordinate.



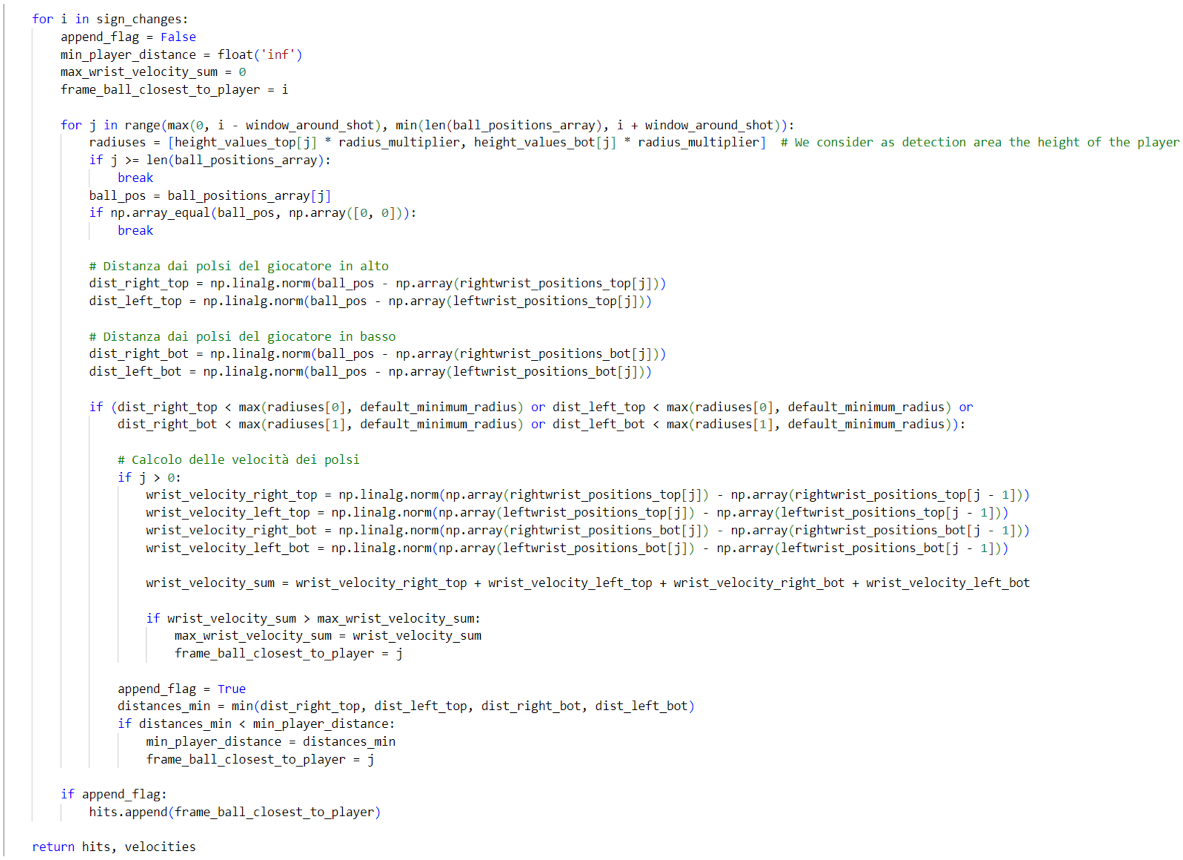
```
        # Estraction of non-zero values to create interpolation function
        non_zero_coords = [(x, y) for x, y in subpositions if (x, y) != (0, 0)]
        #zero_indices = [i for i, point in subpositions if point == (0, 0)]

        x_values = [x for x, _ in non_zero_coords]
        y_values = [y for _, y in non_zero_coords]

        #print("\nMARK 1")
        #for c in non_zero_coords:
        #    print(c)

        # Creation of time axis for the considered interval
        t_axis = []
        #print("\nMARK 2")
        t_inst = 0
        lenghtsubpos = len(subpositions)
        while t_inst < lenghtsubpos:
            if subpositions[t_inst] != (0,0):
                t_axis.append(t_inst)
            t_inst += 1

        zero_indices = []
        t_inst = 0
        while t_inst < lenghtsubpos:
            if subpositions[t_inst] == (0,0):
                zero_indices.append(t_inst)
            t_inst += 1

        #print(f"Lent: {len(t_axis)}")
        #print(f"Lenx: {len(x_values)}")
        #print(f"Leny: {len(y_values)}")

        #Creation of function over x values
        x_interp_func = interp1d(t_axis, x_values, kind='slinear', fill_value='extrapolate') #spline interpolation function

        #Creation of function over y values
        y_interp_func = interp1d(t_axis, y_values, kind='slinear', fill_value='extrapolate') #spline interpolation function

        # Interpolation of missing (zero) values
        for i in zero_indices:
            x_interp_value = int(x_interp_func(i))
            y_interp_value = int(y_interp_func(i))
            coords[i+index-jump] = (x_interp_value, y_interp_value)
            #print(f"Result of interpolation: {x_interp_value}, {y_interp_value}\n")
```

Afterwards, the interpolation loop processes chunks of coordinates and checks whether interpolation is needed or not. Where interpolation is required, linear interpolation is used through the function *interp1d*, which, for each chunk, uses non-zero values to create the interpolation functions over x values and y values. As such, through the previously found interpolation functions, the missing values are filled in and the list of indices where interpolation was applied is returned.



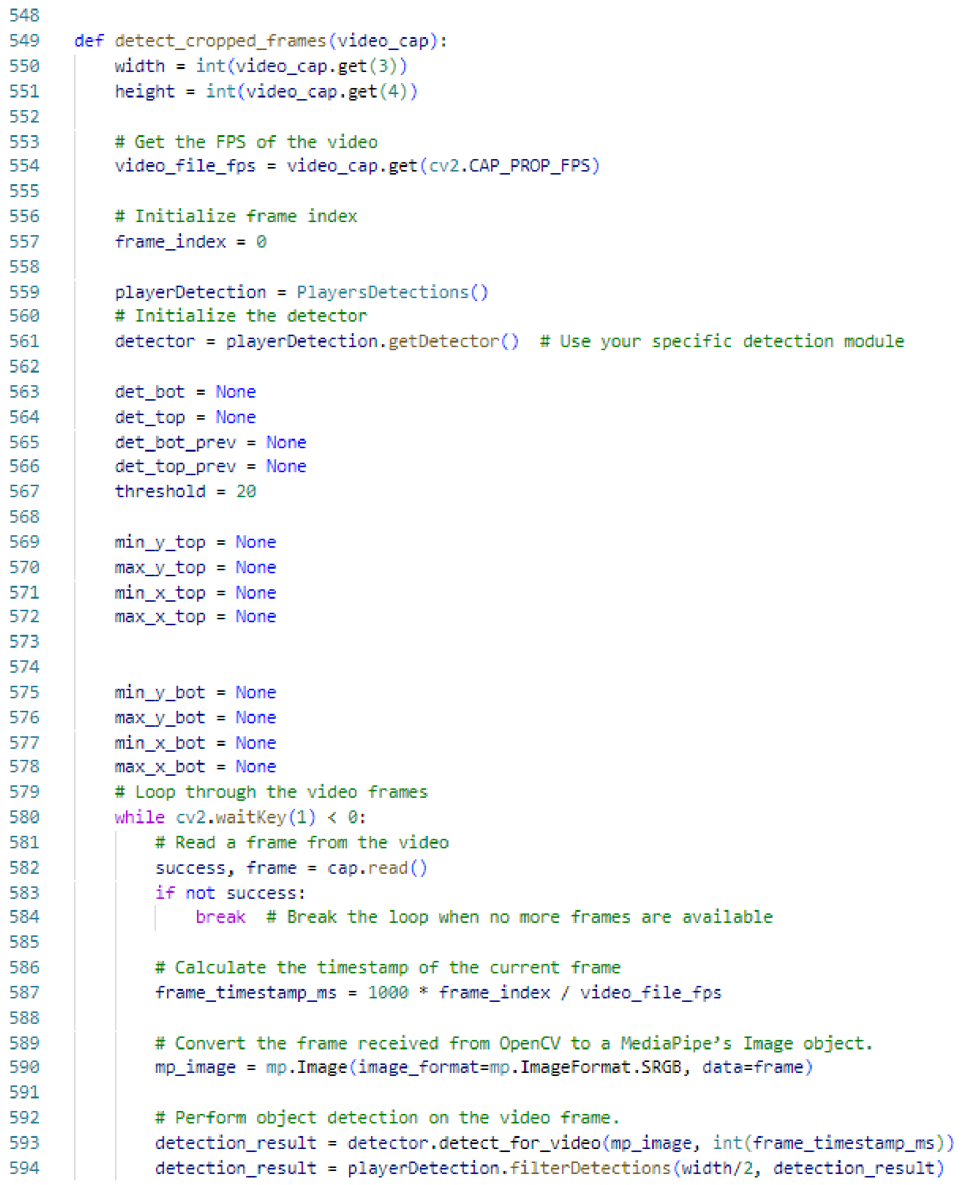
```
def detect_racket_hits(ball_positions, rightwrist_positions_top, leftwrist_positions_top, rightwrist_positions_bot, leftwrist_positions_bot, height_values_top, height_values_bot):
    hits = []
    global vfps
    scaler_freq = vfps / 60
    global res_width
    scaler_res = res_width / 1280 # Scaling the window wrt width or height doesn't matter as long as we're considering a screen ratio of 16:9

    poly_order = 2 # Smoothing fitting degree
    window = int(49*scaler_freq) # Smoothing window (Default for 60FPS is 49)
    if (window %2) != 0: # Savgol filter requires an even window
        window += 1

    ball_positions_array = np.array(ball_positions)
    y_positions = ball_positions_array[:, 1]
    smoothed_y_positions = savgol_filter(y_positions, window, poly_order) # Smoothing to avoid sudden gradient changes due to noise on y_position

    velocities = savgol_filter(y_positions, window, poly_order, deriv=1)  # Vertical derivative on the smoothed function
    accelerations = np.gradient(velocities)

    # Computation of utility graphs
    plotgraph(smoothed_y_positions, "Frame", "y Position", "pC.jpg")
    plotgraph(velocities, "Frame", "y Velocity", "vC.jpg")
    #plotgraph(accelerations, "Frame", "y Accelerations", "aC.jpg")

    sign_changes = np.where(np.diff(np.sign(velocities)))[0] # Derivative changes where we look for racket hits

    window_around_shot = int(20*scaler_freq) # Syncronization window to find the swing of the racket closest to the vertical derivative change (Default for 60FPS is 20)
    default_minimum_radius = int(100*scaler_res) #Pixel minimum player distance considered (Default is 100 px for 720p clips)
    radius_multiplier = 10 # Default multiplier that best fits the correct detection
```

To detect the points in the trajectory of the tennis players when the racket hits the ball, changes in the vertical velocity ($\dot{y}$)   were analyzed through function *detect_Racket_hits* which uses *np.gradient* and then checks whether the sign of the gradient has changed or not. In case of a change, a ball hit is detected which appends the corresponding indices to the hits list. At the end, the list is returned.



```
    for i in sign_changes:
        append_flag = False
        min_player_distance = float('inf')
        max_wrist_velocity_sum = 0
        frame_ball_closest_to_player = i

        for j in range(max(0, i - window_around_shot), min(len(ball_positions_array), i + window_around_shot)):
            radiuses = [height_values_top[j] * radius_multiplier, height_values_bot[j] * radius_multiplier]  # We consider as detection area the height of the player
            if j >= len(ball_positions_array):
                break
            ball_pos = ball_positions_array[j]
            if np.array_equal(ball_pos, np.array([0, 0])):
                break

            # Distanza dai polsi del giocatore in alto
            dist_right_top = np.linalg.norm(ball_pos - np.array(rightwrist_positions_top[j]))
            dist_left_top = np.linalg.norm(ball_pos - np.array(leftwrist_positions_top[j]))

            # Distanza dai polsi del giocatore in basso
            dist_right_bot = np.linalg.norm(ball_pos - np.array(rightwrist_positions_bot[j]))
            dist_left_bot = np.linalg.norm(ball_pos - np.array(leftwrist_positions_bot[j]))

            if (dist_right_top < max(radiuses[0], default_minimum_radius) or dist_left_top < max(radiuses[0], default_minimum_radius) or
                dist_right_bot < max(radiuses[1], default_minimum_radius) or dist_left_bot < max(radiuses[1], default_minimum_radius)):

                # Calcolo delle velocità dei polsi
                if j > 0:
                    wrist_velocity_right_top = np.linalg.norm(np.array(rightwrist_positions_top[j]) - np.array(rightwrist_positions_top[j - 1]))
                    wrist_velocity_left_top = np.linalg.norm(np.array(leftwrist_positions_top[j]) - np.array(leftwrist_positions_top[j - 1]))
                    wrist_velocity_right_bot = np.linalg.norm(np.array(rightwrist_positions_bot[j]) - np.array(rightwrist_positions_bot[j - 1]))
                    wrist_velocity_left_bot = np.linalg.norm(np.array(leftwrist_positions_bot[j]) - np.array(leftwrist_positions_bot[j - 1]))

                    wrist_velocity_sum = wrist_velocity_right_top + wrist_velocity_left_top + wrist_velocity_right_bot + wrist_velocity_left_bot

                    if wrist_velocity_sum > max_wrist_velocity_sum:
                        max_wrist_velocity_sum = wrist_velocity_sum
                        frame_ball_closest_to_player = j

                append_flag = True
                distances_min = min(dist_right_top, dist_left_top, dist_right_bot, dist_left_bot)
                if distances_min < min_player_distance:
                    min_player_distance = distances_min
                    frame_ball_closest_to_player = j

        if append_flag:
            hits.append(frame_ball_closest_to_player)

    return hits, velocities
```

To check the behaviour of the code, see the comments under Chapter 2.

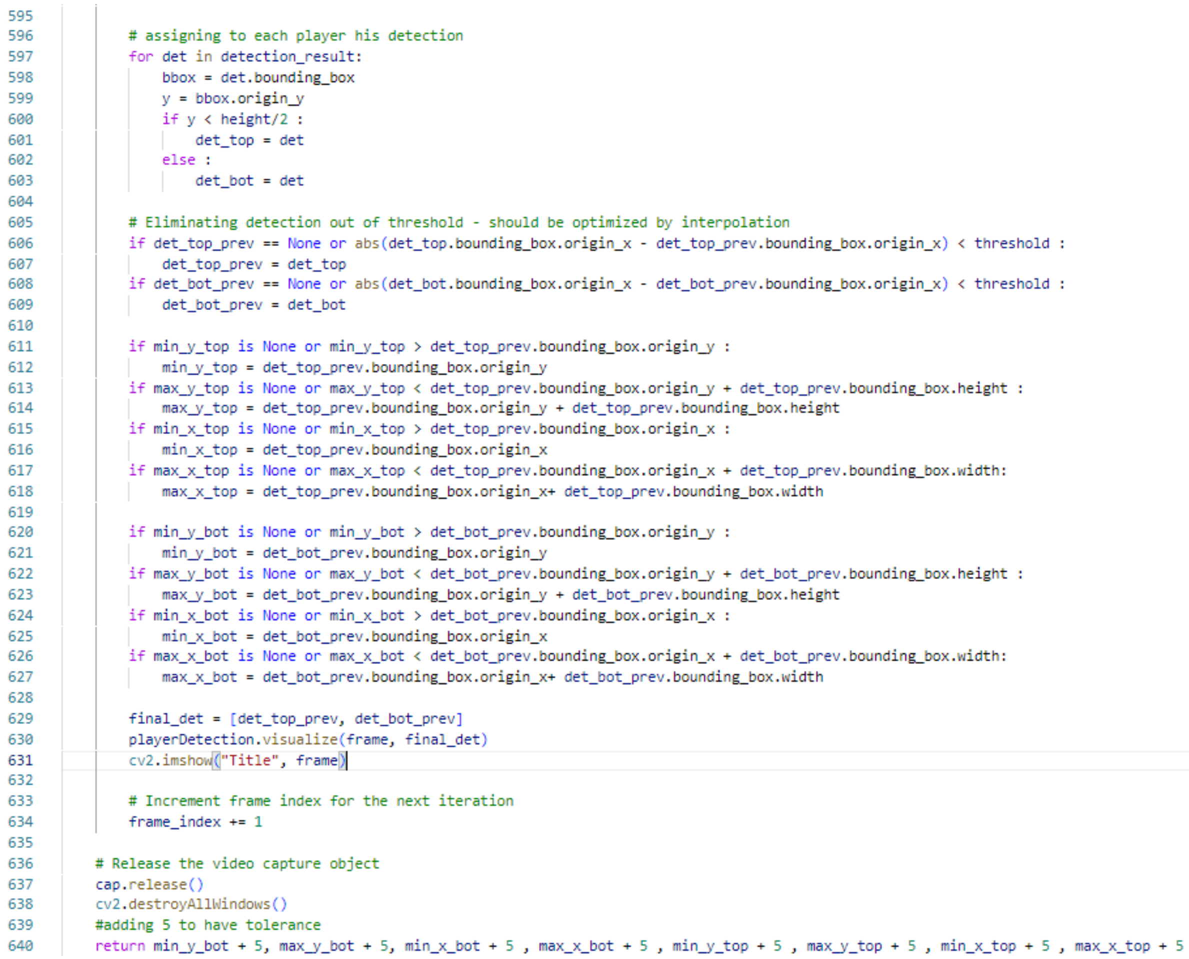


```

def detect_cropped_frames(video_cap):
    width = int(video_cap.get(3))
    height = int(video_cap.get(4))

    # Get the FPS of the video
    video_file_fps = video_cap.get(cv2.CAP_PROP_FPS)

    # Initialize frame index
    frame_index = 0

    playerDetection = PlayersDetections()
    # Initialize the detector
    detector = playerDetection.getDetector()  # Use your specific detection module

    det_bot = None
    det_top = None
    det_bot_prev = None
    det_top_prev = None
    threshold = 20

    min_y_top = None
    max_y_top = None
    min_x_top = None
    max_x_top = None


    min_y_bot = None
    max_y_bot = None
    min_x_bot = None
    max_x_bot = None
    # Loop through the video frames
    while cv2.waitKey(1) < 0:
        # Read a frame from the video
        success, frame = cap.read()
        if not success:
            break  # Break the loop when no more frames are available

        # Calculate the timestamp of the current frame
        frame_timestamp_ms = 1000 * frame_index / video_file_fps

        # Convert the frame received from OpenCV to a MediaPipe’s Image object.
        mp_image = mp.Image(image_format=mp.ImageFormat.SRGB, data=frame)

        # Perform object detection on the video frame.
        detection_result = detector.detect_for_video(mp_image, int(frame_timestamp_ms))
        detection_result = playerDetection.filterDetections(width/2, detection_result)
```

Function *detect_cropped_frames* was defined to detect the bounding boxes of players in the frames and return the cropping coordinates.

It does so by:

1. collecting the width and the height of the video, as well as the frames per second of the video; it also initializes a frame index, as well as a detector using the *PlayersDetections* algorithm previously implemented;
2. using a frame processing loop which reads each frame and detects players’ bounding boxes by:
    a. calculating the time stamp of the current frame;
    b. converting the image read (frame read through OpenCV) to a MediaPipe Image object so that it’s in a suitable format for the detector *mp.Image;*
    c. performing player detection and returning the detection results;



```
        # assigning to each player his detection
        for det in detection_result:
            bbox = det.bounding_box
            y = bbox.origin_y
            if y < height/2 :
                det_top = det
            else :
                det_bot = det

        # Eliminating detection out of threshold - should be optimized by interpolation
        if det_top_prev == None or abs(det_top.bounding_box.origin_x - det_top_prev.bounding_box.origin_x) < threshold :
            det_top_prev = det_top
        if det_bot_prev == None or abs(det_bot.bounding_box.origin_x - det_bot_prev.bounding_box.origin_x) < threshold :
            det_bot_prev = det_bot

        if min_y_top is None or min_y_top > det_top_prev.bounding_box.origin_y :
            min_y_top = det_top_prev.bounding_box.origin_y
        if max_y_top is None or max_y_top < det_top_prev.bounding_box.origin_y + det_top_prev.bounding_box.height :
            max_y_top = det_top_prev.bounding_box.origin_y + det_top_prev.bounding_box.height
        if min_x_top is None or min_x_top > det_top_prev.bounding_box.origin_x :
            min_x_top = det_top_prev.bounding_box.origin_x
        if max_x_top is None or max_x_top < det_top_prev.bounding_box.origin_x + det_top_prev.bounding_box.width:
            max_x_top = det_top_prev.bounding_box.origin_x+ det_top_prev.bounding_box.width

        if min_y_bot is None or min_y_bot > det_bot_prev.bounding_box.origin_y :
            min_y_bot = det_bot_prev.bounding_box.origin_y
        if max_y_bot is None or max_y_bot < det_bot_prev.bounding_box.origin_y + det_bot_prev.bounding_box.height :
            max_y_bot = det_bot_prev.bounding_box.origin_y + det_bot_prev.bounding_box.height
        if min_x_bot is None or min_x_bot > det_bot_prev.bounding_box.origin_x :
            min_x_bot = det_bot_prev.bounding_box.origin_x
        if max_x_bot is None or max_x_bot < det_bot_prev.bounding_box.origin_x + det_bot_prev.bounding_box.width:
            max_x_bot = det_bot_prev.bounding_box.origin_x+ det_bot_prev.bounding_box.width

        final_det = [det_top_prev, det_bot_prev]
        playerDetection.visualize(frame, final_det)
        cv2.imshow("Title", frame)

        # Increment frame index for the next iteration
        frame_index += 1

    # Release the video capture object
    cap.release()
    cv2.destroyAllWindows()
    #adding 5 to have tolerance
    return min_y_bot + 5, max_y_bot + 5, min_x_bot + 5 , max_x_bot + 5 , min_y_top + 5 , max_y_top + 5 , min_x_top + 5 , max_x_top + 5
```

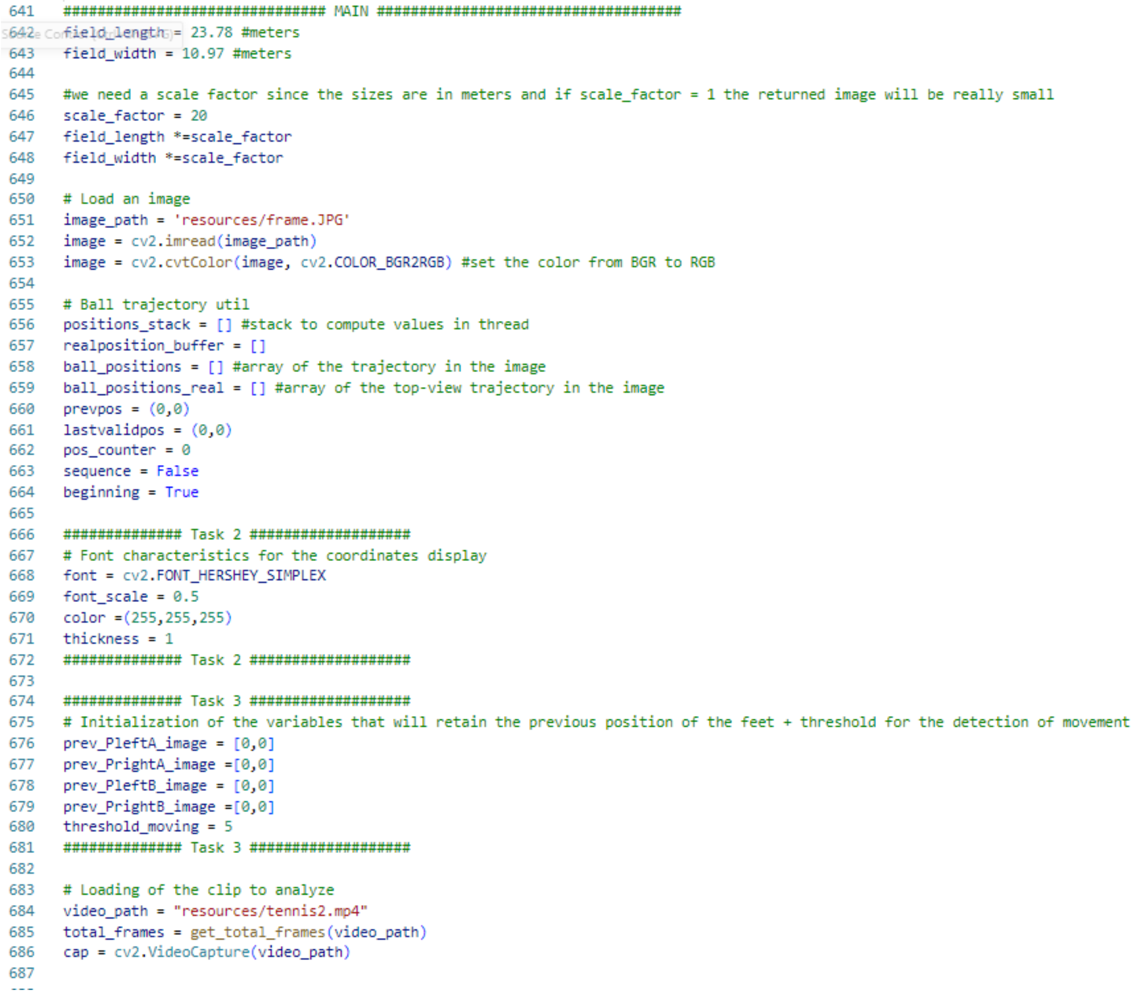
d. filtering the detections based on their position relative to the middle of the frame (it separates the top and bottom players detections); the bottom and upper detections are assigned based on their y coordinate;

e. updating previous detections based on whether the current detection is within a threshold (close distance) from the previous one or if there was no previous detection;

f. determining the cropping coordinates based on the detected bounding boxes o the players (if the current Y coordinate < previous Y or if the current detection's bottom edge (Y + height) > previous Y then update Y AND if the current X coordinate < previous X or if the current detection's bottom edge (X + width) > previous X then update X).

3. combining the final detections for visualization and drawing the bounding boxes on the frame + displaying the frame with detections in a window and moving to the next frame;

At the end of the function, the calculated coordinates for cropping frames around the players are returned.



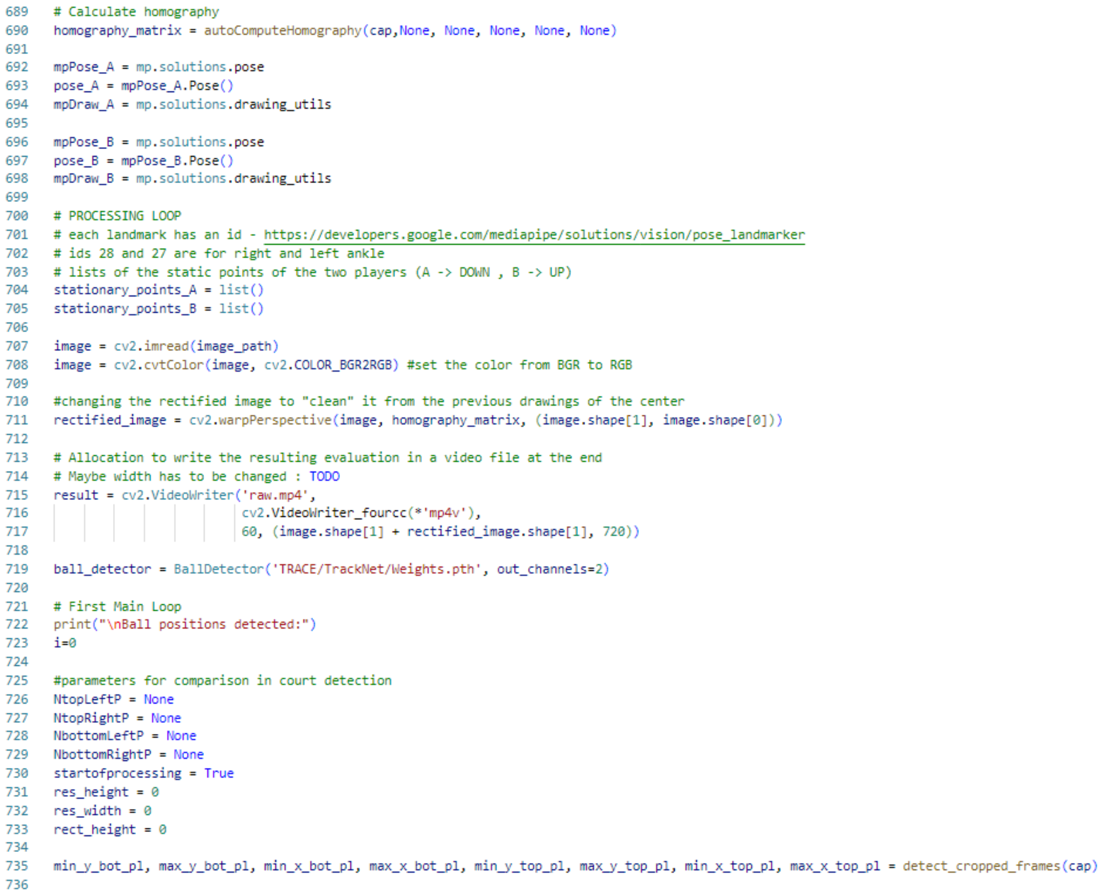
```
############################### MAIN ####################################
field_length = 23.78 #meters
field_width = 10.97 #meters

#we need a scale factor since the sizes are in meters and if scale_factor = 1 the returned image will be really small
scale_factor = 20
field_length *=scale_factor
field_width *=scale_factor

# Load an image
image_path = 'resources/frame.JPG'
image = cv2.imread(image_path)
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB) #set the color from BGR to RGB

# Ball trajectory util
positions_stack = [] #stack to compute values in thread
realposition_buffer = []
ball_positions = [] #array of the trajectory in the image
ball_positions_real = [] #array of the top-view trajectory in the image
prevpos = (0,0)
lastvalidpos = (0,0)
pos_counter = 0
sequence = False
beginning = True

############## Task 2 ###################
# Font characteristics for the coordinates display
font = cv2.FONT_HERSHEY_SIMPLEX
font_scale = 0.5
color =(255,255,255)
thickness = 1
############## Task 2 ###################

############## Task 3 ###################
# Initialization of the variables that will retain the previous position of the feet + threshold for the detection of movement
prev_PleftA_image = [0,0]
prev_PrightA_image =[0,0]
prev_PleftB_image = [0,0]
prev_PrightB_image =[0,0]
threshold_moving = 5
############## Task 3 ###################

# Loading of the clip to analyze
video_path = "resources/tennis2.mp4"
total_frames = get_total_frames(video_path)
cap = cv2.VideoCapture(video_path)

```

Arriving at the main function, the tennis court real dimensions are initialized to then become scaled with a factor of 20 for visualization.

An image of the tennis court is loaded and converted to RGB color space. Afterwards, the initial video frame and ball trajectory arrays are set up.

Additionally, for displaying purposes, font characteristics are set along with the threshold used for movement detection and with the initialization of the variables that will retain the previous position of the feet.

The main function continues to be made up of code responsible for loading up the video to be analyzed. As seen, function *get_total_frames* is called.



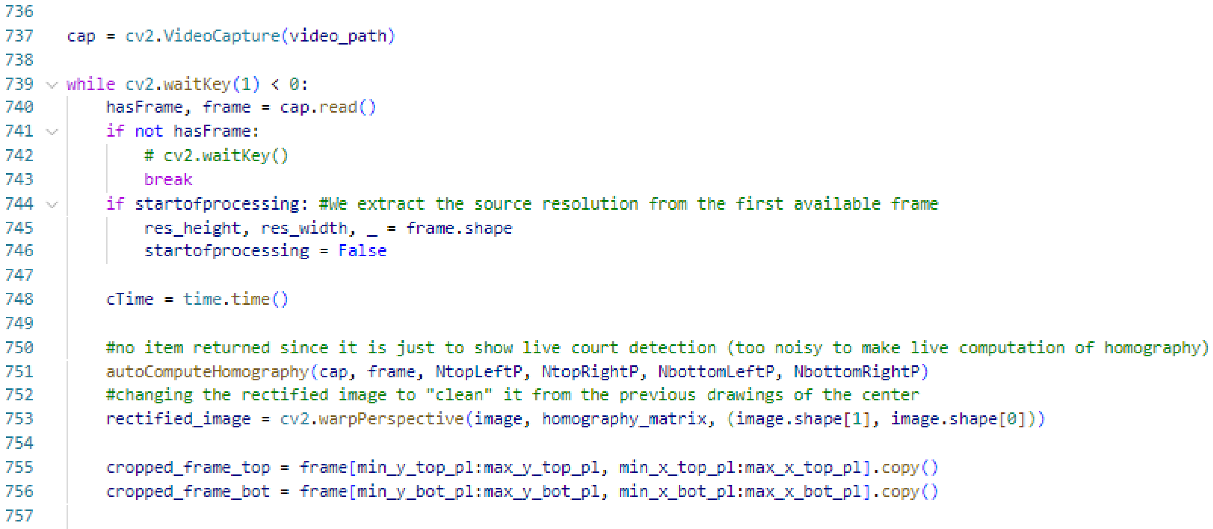
```
689  # Calculate homography
690  homography_matrix = autoComputeHomography(cap,None, None, None, None, None)
691
692  mpPose_A = mp.solutions.pose
693  pose_A = mpPose_A.Pose()
694  mpDraw_A = mp.solutions.drawing_utils
695
696  mpPose_B = mp.solutions.pose
697  pose_B = mpPose_B.Pose()
698  mpDraw_B = mp.solutions.drawing_utils
699
700  # PROCESSING LOOP
701  # each landmark has an id - https://developers.google.com/mediapipe/solutions/vision/pose_landmarker
702  # ids 28 and 27 are for right and left ankle
703  # lists of the static points of the two players (A -> DOWN , B -> UP)
704  stationary_points_A = list()
705  stationary_points_B = list()
706
707  image = cv2.imread(image_path)
708  image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB) #set the color from BGR to RGB
709
710  #changing the rectified image to "clean" it from the previous drawings of the center
711  rectified_image = cv2.warpPerspective(image, homography_matrix, (image.shape[1], image.shape[0]))
712
713  # Allocation to write the resulting evaluation in a video file at the end
714  # Maybe width has to be changed : TODO
715  result = cv2.VideoWriter('raw.mp4',
716                           cv2.VideoWriter_fourcc(*'mp4v'),
717                           60, (image.shape[1] + rectified_image.shape[1], 720))
718
719  ball_detector = BallDetector('TRACE/TrackNet/Weights.pth', out_channels=2)
720
721  # First Main Loop
722  print("\nBall positions detected:")
723  i=0
724
725  #parameters for comparison in court detection
726  NtopLeftP = None
727  NtopRightP = None
728  NbottomLeftP = None
729  NbottomRightP = None
730  startofprocessing = True
731  res_height = 0
732  res_width = 0
733  rect_height = 0
734
735  min_y_bot_pl, max_y_bot_pl, min_x_bot_pl, max_x_bot_pl, min_y_top_pl, max_y_top_pl, min_x_top_pl, max_x_top_pl = detect_cropped_frames(cap)
736
```

Subsequently the homography matrix is calculated for perspective transforming.

Pose detection models, mpPose_A, pose_A, mpPose_B and pose_B are initialized along with drawing utilities for detecting player's poses.

Continuing, the main function is also responsible for:

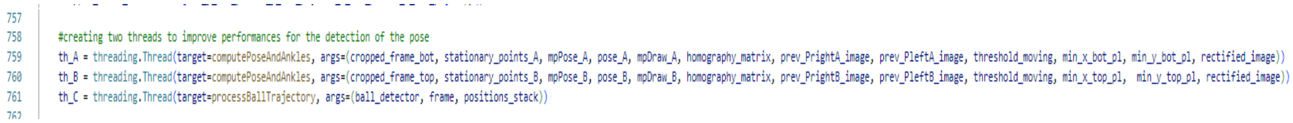
- initializing the lists containing the stationary points of each tennis player;
- reading the image with the tennis court to then be converted to RGB color space;
- rectifying the tennis court image using the homography matrix;
- initializing the video writer;
- loading the TRACE model responsible for ball tracking;
- initializing the parameters for court detection;
- detecting the cropped frames;
- reloading the video for processing (line 737).



```
cap = cv2.VideoCapture(video_path)

while cv2.waitKey(1) < 0:
    hasFrame, frame = cap.read()
    if not hasFrame:
        # cv2.waitKey()
        break
    if startofprocessing: #We extract the source resolution from the first available frame
        res_height, res_width, _ = frame.shape
        startofprocessing = False

    cTime = time.time()

    #no item returned since it is just to show live court detection (too noisy to make live computation of homography)
    autoComputeHomography(cap, frame, NtopLeftP, NtopRightP, NbottomLeftP, NbottomRightP)
    #changing the rectified image to "clean" it from the previous drawings of the center
    rectified_image = cv2.warpPerspective(image, homography_matrix, (image.shape[1], image.shape[0]))

    cropped_frame_top = frame[min_y_top_pl:max_y_top_pl, min_x_top_pl:max_x_top_pl].copy()
    cropped_frame_bot = frame[min_y_bot_pl:max_y_bot_pl, min_x_bot_pl:max_x_bot_pl].copy()

```

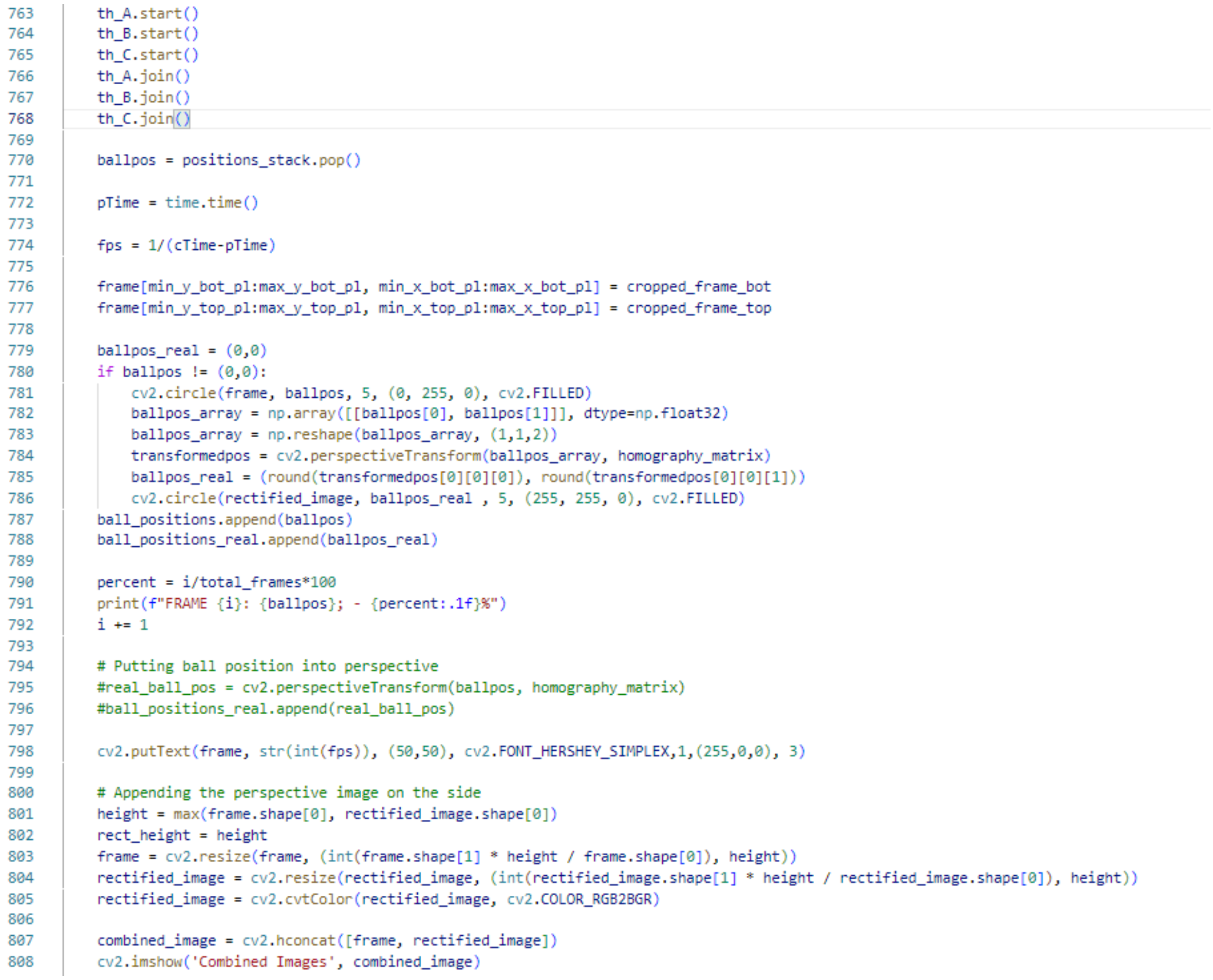
Arriving at the main processing loop, the algorithm:

1. reads a frame from the video capture object and extracts the source resolution from the first one available;
2. computes the homography matrix for the current frame to adjust the perspective through the previously defined function autoComputeHomography;
3. crops frames around the detected bounding boxes for the top and bottom players;



```

    #creating two threads to improve performances for the detection of the pose
    th_A = threading.Thread(target=computePoseAndAnkles, args=(cropped_frame_bot, stationary_points_A, mpPose_A, pose_A, mpDraw_A, homography_matrix, prev_PrightA_image, prev_PleftA_image, threshold_moving, min_x_bot_pl, min_y_bot_pl, rectified_image))
    th_B = threading.Thread(target=computePoseAndAnkles, args=(cropped_frame_top, stationary_points_B, mpPose_B, pose_B, mpDraw_B, homography_matrix, prev_PrightB_image, prev_PleftB_image, threshold_moving, min_x_top_pl,  min_y_top_pl, rectified_image))
    th_C = threading.Thread(target=processBallTrajectory, args=(ball_detector, frame, positions_stack))

```

4. creates threads to improve the performance of the pose detection and ball trajectory processing;

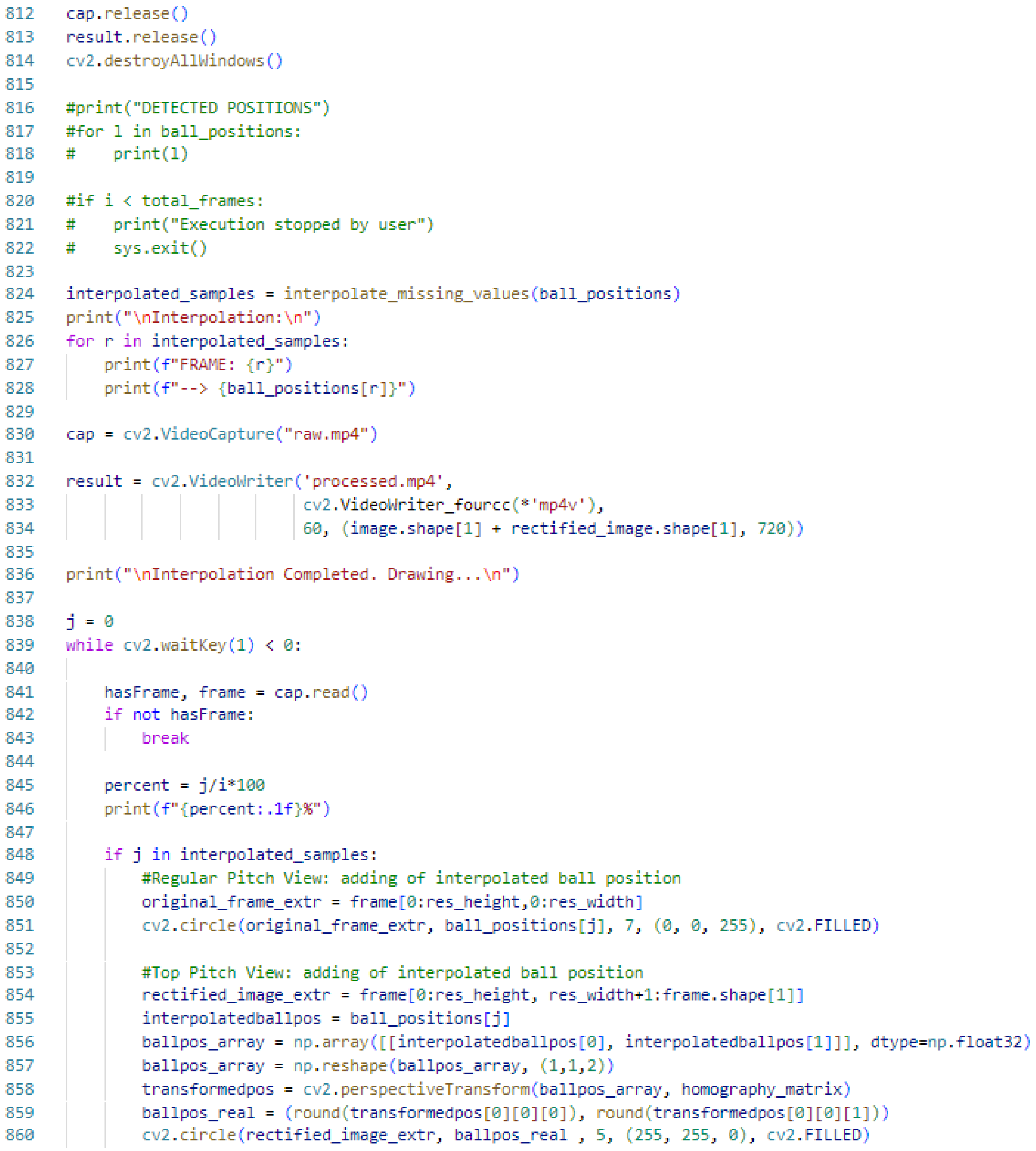


```
763     th_A.start()
764     th_B.start()
765     th_C.start()
766     th_A.join()
767     th_B.join()
768     th_C.join()
769 
770     ballpos = positions_stack.pop()
771 
772     pTime = time.time()
773 
774     fps = 1/(cTime-pTime)
775 
776     frame[min_y_bot_pl:max_y_bot_pl, min_x_bot_pl:max_x_bot_pl] = cropped_frame_bot
777     frame[min_y_top_pl:max_y_top_pl, min_x_top_pl:max_x_top_pl] = cropped_frame_top
778 
779     ballpos_real = (0,0)
780     if ballpos != (0,0):
781         cv2.circle(frame, ballpos, 5, (0, 255, 0), cv2.FILLED)
782         ballpos_array = np.array([[ballpos[0], ballpos[1]]], dtype=np.float32)
783         ballpos_array = np.reshape(ballpos_array, (1,1,2))
784         transformedpos = cv2.perspectiveTransform(ballpos_array, homography_matrix)
785         ballpos_real = (round(transformedpos[0][0][0]), round(transformedpos[0][0][1]))
786         cv2.circle(rectified_image, ballpos_real , 5, (255, 255, 0), cv2.FILLED)
787     ball_positions.append(ballpos)
788     ball_positions_real.append(ballpos_real)
789 
790     percent = i/total_frames*100
791     print(f"FRAME {i}: {ballpos}; - {percent:.1f}%")
792     i += 1
793 
794     # Putting ball position into perspective
795     #real_ball_pos = cv2.perspectiveTransform(ballpos, homography_matrix)
796     #ball_positions_real.append(real_ball_pos)
797 
798     cv2.putText(frame, str(int(fps)), (50,50), cv2.FONT_HERSHEY_SIMPLEX,1,(255,0,0), 3)
799 
800     # Appending the perspective image on the side
801     height = max(frame.shape[0], rectified_image.shape[0])
802     rect_height = height
803     frame = cv2.resize(frame, (int(frame.shape[1] * height / frame.shape[0]), height))
804     rectified_image = cv2.resize(rectified_image, (int(rectified_image.shape[1] * height / rectified_image.shape[0]), height))
805     rectified_image = cv2.cvtColor(rectified_image, cv2.COLOR_RGB2BGR)
806 
807     combined_image = cv2.hconcat([frame, rectified_image])
808     cv2.imshow('Combined Images', combined_image)
```

Further down the code lines, the algorithm starts and joins the threads to ensure concurrent execution and completion.

The ball position in both the original and the rectified frames is updated while the frames per second are calculated to monitor the processing speed. Subsequently, the frames are updated with the cropped regions and ball positions.

Finally, the computed results are saved, while the video resources are released. (line 812-814).



```
812    cap.release()
813    result.release()
814    cv2.destroyAllWindows()
815
816    #print("DETECTED POSITIONS")
817    #for l in ball_positions:
818    #    print(l)
819
820    #if i < total_frames:
821    #    print("Execution stopped by user")
822    #    sys.exit()
823
824    interpolated_samples = interpolate_missing_values(ball_positions)
825    print("\nInterpolation:\n")
826    for r in interpolated_samples:
827        print(f"FRAME: {r}")
828        print(f"--> {ball_positions[r]}")
829
830    cap = cv2.VideoCapture("raw.mp4")
831
832    result = cv2.VideoWriter('processed.mp4',
833                             cv2.VideoWriter_fourcc(*'mp4v'),
834                             60, (image.shape[1] + rectified_image.shape[1], 720))
835
836    print("\nInterpolation Completed. Drawing...\n")
837
838    j = 0
839    while cv2.waitKey(1) < 0:
840
841        hasFrame, frame = cap.read()
842        if not hasFrame:
843            break
844
845        percent = j/i*100
846        print(f"{percent:.1f}%")
847
848        if j in interpolated_samples:
849            #Regular Pitch View: adding of interpolated ball position
850            original_frame_extr = frame[0:res_height,0:res_width]
851            cv2.circle(original_frame_extr, ball_positions[j], 7, (0, 0, 255), cv2.FILLED)
852
853            #Top Pitch View: adding of interpolated ball position
854            rectified_image_extr = frame[0:res_height, res_width+1:frame.shape[1]]
855            interpolatedballpos = ball_positions[j]
856            ballpos_array = np.array([[interpolatedballpos[0], interpolatedballpos[1]]], dtype=np.float32)
857            ballpos_array = np.reshape(ballpos_array, (1,1,2))
858            transformedpos = cv2.perspectiveTransform(ballpos_array, homography_matrix)
859            ballpos_real = (round(transformedpos[0][0][0]), round(transformedpos[0][0][1]))
860            cv2.circle(rectified_image_extr, ballpos_real , 5, (255, 255, 0), cv2.FILLED)
```

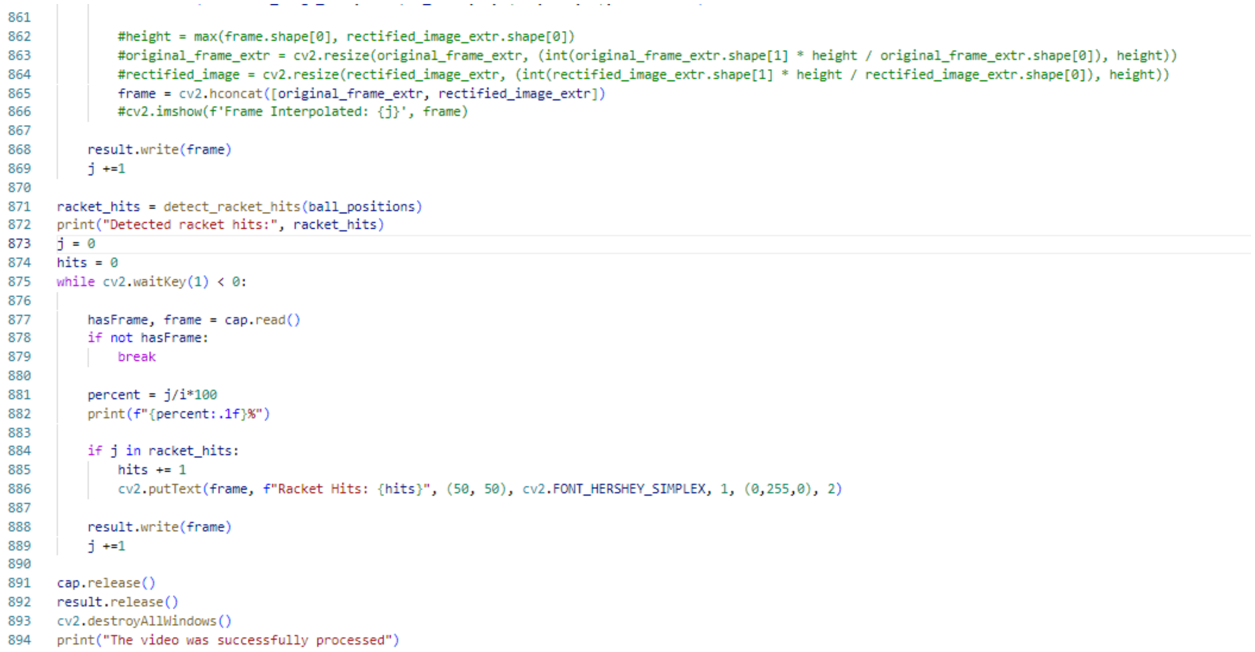
From line 824, the algorithm takes advantage of the previously defined important function *interpolate_missing_values* filling in any gaps in the detected ball positions, then printing the frame number and their corresponding interpolated ball positions.

Later, the processed video (*raw.mp4*) is reopened for reading and a new video writer is initialized for the final output (*processed.mp4*).

Thus, the algorithm reads frames from the video and:

- calculates and prints the percentage of completion;

- adds circles to mark the interpolated ball positions on both the original and rectified views;
- combines the original and rectified frames side by side (line 865);



```
861
862         #height = max(frame.shape[0], rectified_image_extr.shape[0])
863         #original_frame_extr = cv2.resize(original_frame_extr, (int(original_frame_extr.shape[1] * height / original_frame_extr.shape[0]), height))
864         #rectified_image = cv2.resize(rectified_image_extr, (int(rectified_image_extr.shape[1] * height / rectified_image_extr.shape[0]), height))
865         frame = cv2.hconcat([original_frame_extr, rectified_image_extr])
866         #cv2.imshow(f'Frame Interpolated: {j}', frame)
867
868     result.write(frame)
869     j +=1
870
871 racket_hits = detect_racket_hits(ball_positions)
872 print("Detected racket hits:", racket_hits)
873 j = 0
874 hits = 0
875 while cv2.waitKey(1) < 0:
876
877     hasFrame, frame = cap.read()
878     if not hasFrame:
879         break
880
881     percent = j/i*100
882     print(f"{percent:.1f}%")
883
884     if j in racket_hits:
885         hits += 1
886         cv2.putText(frame, f"Racket Hits: {hits}", (50, 50), cv2.FONT_HERSHEY_SIMPLEX, 1, (0,255,0), 2)
887
888     result.write(frame)
889     j +=1
890
891 cap.release()
892 result.release()
893 cv2.destroyAllWindows()
894 print("The video was successfully processed")
```

- writes the annotated frame to the result video;

In the last part of the code, the function *detect_racket_hits* identifies the frames where racket hits occur to which text indicating the count of racket hits is added.

The last lines of code are used to release the video resources and close all OpenCV windows, printing the message: “The video was successfully processed”.

4 Result Analysis

4.1 Output

The implemented solution provides several clips as intermediate memory allocation to append computation results on, specifically:

- A *raw.mp4* clip that first detects Players, Ball, computes Homography and computes statistics on players movement
- A *processed.mp4* clip that includes the complete ball trajectory detected through interpolation on top of
- A final *processed_winfo.mp4* clip that includes Racket Hit and Average speed between hits

Along with these files the solution computes:

- A .jpg graph of the processing performance, ball vertical position and speed per pixel is saved through the built-in *plotgraph* function
- A real-time telemetry of the processing along with percentage of completion and statistics on processing times in the commandline

4.2 Homography computation

As we can see in the following images, the automatic computation of the homography worked well on almost all tournaments:

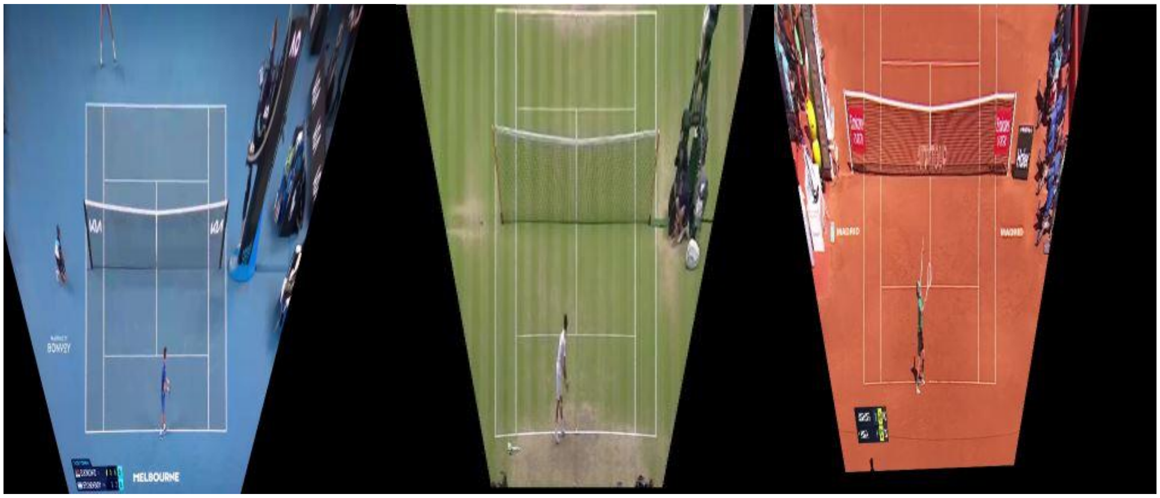


Figure 4.1 Comparison between the rectifications of the different types of tennis fields using the computed homography

But as we can see in the following images, the presence of external factors such as brand logos and shade, make it difficult to obtain an accurate homography from the field.

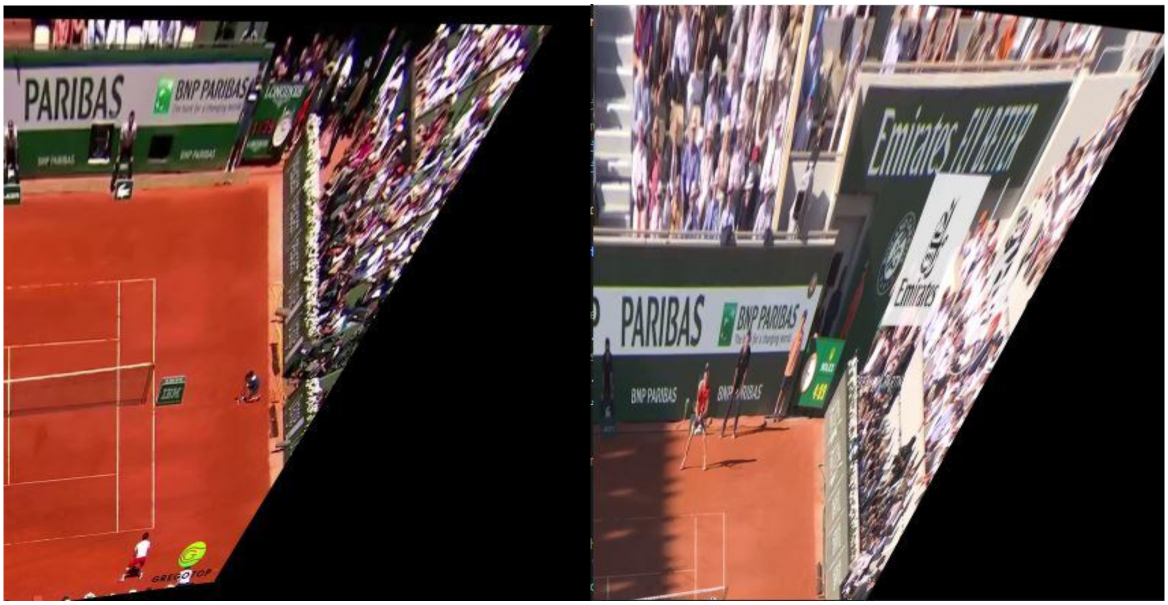


Figure 4.2 Homography affected by external elements - Roland Garros

Moreover, we have focused on the Australian Open tournament, in which there are few external elements that could affect the detection of the players or the computation of the homography.

In fact, in the Australian Open there are just two referees at the two sides of the field. In addition, the wall behind the top player is completely black, without elements that could affect the accuracy of our models. Instead, it is working in our favor since the high color contrast between the wall and the player makes it easier to detect him.

On the other hand, in tournaments such as Roland Garros in which the color of the field is similar to the skin color of the players, the accuracy of the algorithm could be affected and would create noisier detections, which corresponds to an inaccurate detection of the players.

In both Wimbledon and Roland Garros tournaments, due to the presence of the third referee at the center of the field, the human detection model and human pose estimation model are wrongly detecting the top player. Moreover, in Roland Garros there are a lot of elements on the wall behind the top player that are making its detection even more difficult.

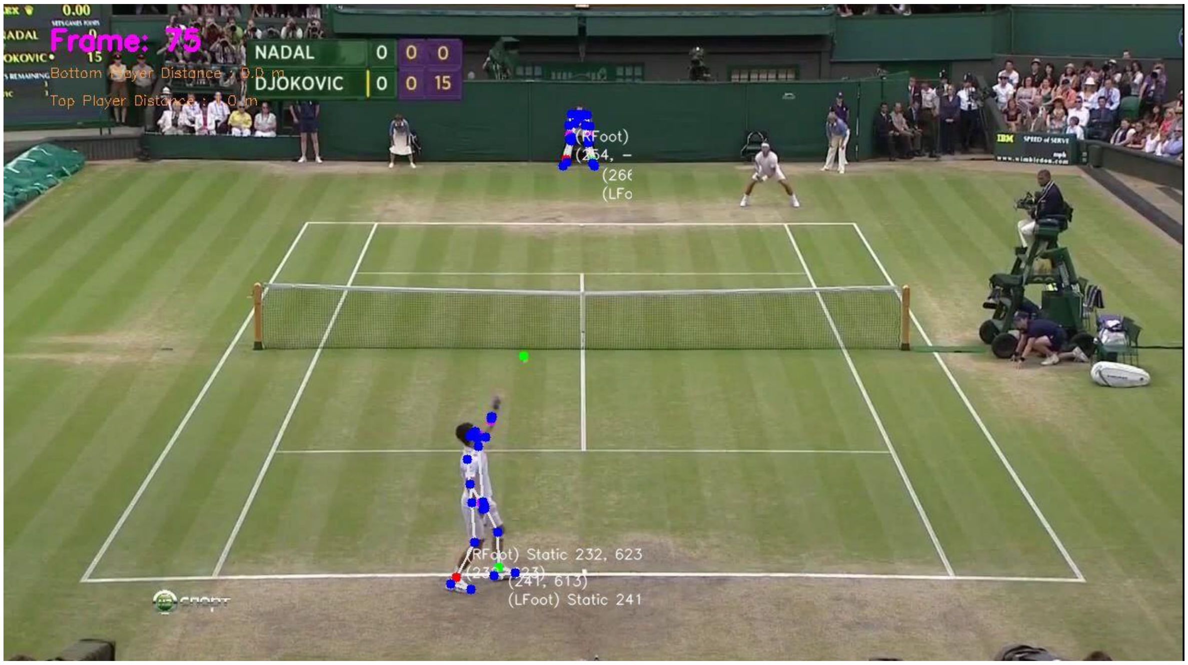


Figure 4.3 Upper Tennis player detection miss - referee detection

Finally, one element that we could not solve was the jump of a player: in fact, the algorithm detects a jump as if the player is going forward and the trajectory is modified accordingly.

For these reasons, possible improvements could be applied to:

- Dynamical computation of the players' human pose through frame pre-processing
- Error correction and improvement in ball trajectory post-detection
- Racket Hit detection through 3D approximation of the ball and top-view estimation

Most of the challenges have been overcome, but the real difficulty was represented by the tennis ball position tracking and filtering to detect racket hits accurately. Statistics were computed on player movement, distance covered during a single tennis point and average speed between hits, but further improvement and more-relevant data can be obtained through 3D estimation of the ball trajectory, which is still an open issue in single-camera setups and needs some sort of approximation to be obtained.

4.3 Ball trajectory computation

In terms of Ball Computation, we could see straight away that TrackNet performed better than YOLOv8 for the considerations exposed in chapter 2.4.

By considering the differences, YOLOv8 never showed a higher probability of ball detection than 0.5 in the considered clip, while the trajectory detected by TrackNet showed a fidelity of above 0.5 (which was actually set as the threshold of the model by the TRACE interface) on around 80% of the detected ball positions, as we inherit from the ratio between total frames and frames where interpolation was not performed.

Depending on framerate and image quality, some misdetection has been spotted in some tests where scenario was far from ideal (e.g. shoes/wristband being of the same color palette of the player), but for isolated frames. This could be avoided by fine-tuning the algorithm based on the processed clip, that can support different error avoidance thresholds in the *processBallTrajectory* function.

We consider the resulting final trajectory to be flawless in ideal cases, like the considered clip of the Australian Open that has a high contrast between environment and ball color.

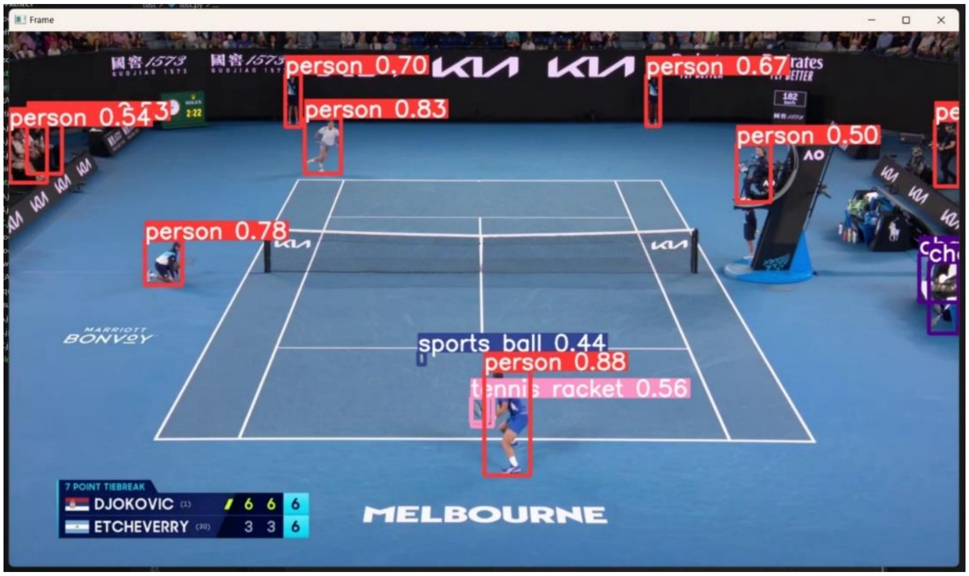


Figure 4.4 YOLOv8 poor performance in ball detection

4.4 Racket Hit computation and statistics

In terms of Racket hit detection, we consider having found a highly effective solution that doesn't require to fully estimate the 3D trajectory of the ball in the considered scene. Such task would have allowed a straight-forward computation of the Racket Hit by considering the top-view of the field and the projected ball trajectory onto it, and the gradient changes on the trajectory observed. Since this has been proven to be a hard task to be performed in single-camera setups, our solution actually managed to get past this issue and find Racket Hit detection through mathematical and signal processing methods.

On ideal cases, our method has proven to be able to detect with high accuracy gradient changes on the original clip and sync them to the actual movement of the hands performed by players, and has performed flawlessly in any ball trajectory detected with a reasonable amount of ball position noise.

On the other hand, the statistics chosen were relative to both player movement and distance covered through the considered homography calculation, detecting the distance covered by the players over the considered clip. Such statistics have been shown to be realistic through any video length presented when homography is correctly detected. Average ball speed was included as one of the possible statistics to be computed over ball trajectory, specifically referring to the average speed observed on each 2-hit ball exchange. Although such statistics isn't related to the well-known hawkeye serve speed, which requires a multiple high-precision camera set to be computed, average ball speed data can be useful in terms of exchange speed, reaction time of the players and energy involved in each shot by players, considering environmental variables. For this reason, serve speed and average ball speed after serve show a significant gap in terms of value observed, as visible in the following image, but they are both meaningful from an analytical point of view. Average speed statistics have shown us to be vulnerable to sudden and high gradient changes due to noise on ball detection.

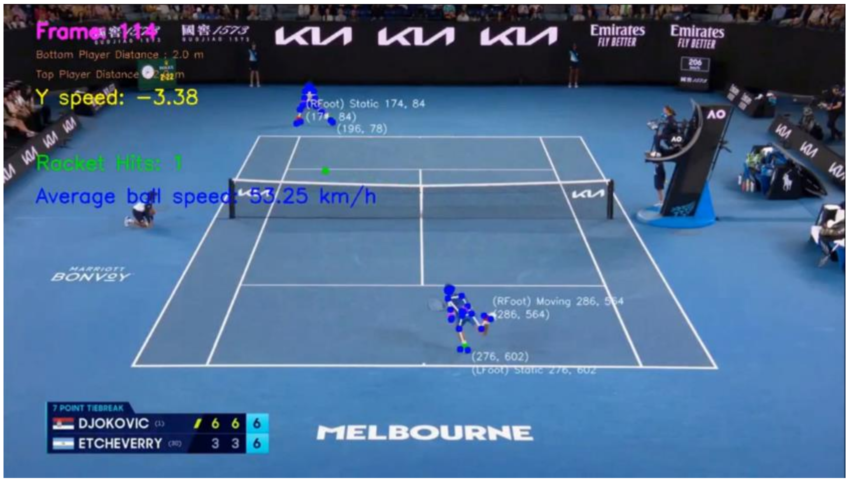
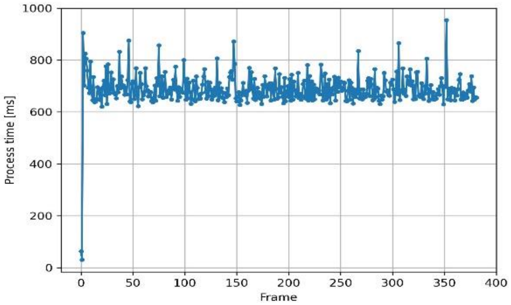


Figure 4.5 Computed match statistics during the video analysis

4.5 Processing Time and Hardware Performance

Through built-in telemetry function and data structures we were able to observe processing times. For performance evaluation, we selected a restricted 6-seconds clip from the main one used to output result, obtaining the following statistics:



```
Total processing time: 296.09 s
Average processing time: 775.10 ms/frame
Max: 1042.95 ms - Min: 31.03 ms
```

Figure 4.6 Computational statistics of the Video analysis

5 Conclusion

Overall, trajectory tracking of tennis players through a monocular video taken by a single static camera could potentially become difficult due to several factors such as low-quality camera and motion blurs caused by low framerates. However, we have tried to get past these obstacles by applying different models and methods of human pose estimation. Hence, our solution is modeled on the possible clip parameters, directly being able to suit different formats and resolutions.

Our solution was able to satisfy all main requests of the project specifications, merging different implementations from the current state of the art, along with the testing and selection from different implementations coming both from theoretical ideas and both from existing solutions.

The main challenges presented came out to be:

- Dynamical selection of the frame areas where to estimate the human pose
- Automatic detection of movement associated to players’ feet
- Ball Trajectory interpolation and Filtering
- Racket Hit detection through ball trajectory
- Automatic computation of the homography from field to image

By testing other videos of other tournaments, it came out that the quality of the video, as well as its frame rate, were significant factors affecting the quality of the results. In fact, the video on which we focused had an HD resolution and a frame rate of 60 frames per second. Other videos with lower resolution, and mainly lower frame rates, were performing worse, but the solution was implemented based on empiric tests on the optimal parameters for 720p 60FPS sources and built to adapt such parameters to different resolutions and framerates.

6 References

[1] Mingxing Tan, Ruoming Pang and Quoc V. Le, "EfficientDet: Scalable and Efficient Object Detection," 2020.

[2] Yu-Chuan Huang, I-No Liao, Ching-Hsuan Chen, Tsì-Uí İk and Wen-Chih Peng, "TrackNet: A Deep Learning Network for Tracking High-speed and Tiny Objects in Sports Applications," Hsinchu, 2019.

[3] T. Sijo, "Impact of image and video visual analysis in sports," *International Journal of Physiology, Nutrition and Physical Education,* vol. 4, no. 1, pp. 1294-1296, 2019.

[4] Google and other contributors, "Pose landmark detection guide," Google Ireland Limited, 2024. [Online]. Available: https://ai.google.dev/edge/mediapipe/solutions/vision/pose_landmarker. [Accessed 04 05 2024].

[5] C. Perin, R. Vuillemot, C. D. Stolper, J. T. Stasko, J. Wood and S. Carpendale, "State of the Art of Sports Data Visualization," *Computer Graphics Forum,* vol. 37, no. 3, 2018.