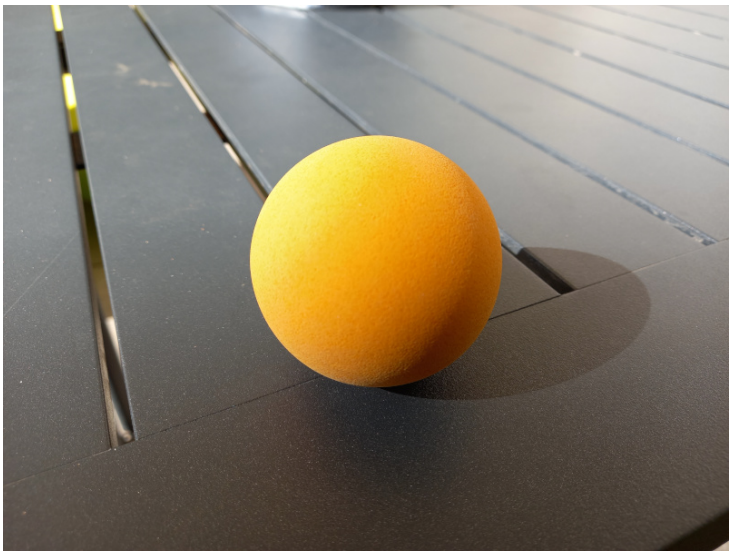
Perception - Tracking de balle avec OpenCV



Auteur : *MUSUMECI Lucas*

Date : *29 octobre 2025*

MEA

POLYTECH©

Polytech Montpellier

Git: git@

Table des matières

I – Identification de la balle sur une image fixe

I.1 Extraction de la couleur HSV de la balle

Avant de commencer notre traitement de l'image, nous avons besoin de récupérer une couleur de référence qui nous permettra d'identifier les pixels appartenant à la balle s'ils sont "semblables" à la couleur de référence. Plutôt que de travailler dans l'espace de couleur RGB, nous allons travailler, pour la comparaison des pixels, dans l'espace HSV, car cela nous permet d'isoler la chrominance, la saturation et la luminosité.

Nous allons donc demander à l'utilisateur de cliquer sur un pixel appartenant à la balle (idéalement, pas dans une zone trop sombre ni trop lumineuse). Avec un callback, nous allons pouvoir récupérer les coordonnées du pixel cliqué et extraire tous les pixels de l'image en HSV dans un carré de côté $2R$ centré sur le pixel cliqué. À partir de cet ensemble de pixels, nous allons obtenir notre pixel de référence à partir de la médiane de chacune des composantes HSV des pixels de la zone.

```python
def get_color_HSV(event, x, y, flags, param):
    if event == cv.EVENT_LBUTTONDOWN:
        img, R = param
        # Conversion de l'image en HSV
        imgHSV = cv.cvtColor(img.copy(), cv.COLOR_BGR2HSV)
        global color_HSV
        global color_HSV_original
        global color_selected

        # Extraction d'une sous partie de l'image autour du point cliqué
        roi = imgHSV[y-R:y+R, x-R:x+R]
        color_HSV = np.median(roi, axis=(0, 1))
        color_HSV_original = color_HSV.copy()

        color_selected = True
        print("Selected color (HSV): ", color_HSV)
```

Code I.1 – Code du callback qui calcule le pixel de référence

I.2 Obtention du masque de l'image par comparaison avec la couleur de référence

Pour obtenir le masque de l'image, nous pourrions parcourir chaque pixel de l'image avec une double boucle for et le comparer à notre pixel de référence avec une certaine tolérance sur chacune des composantes. Néanmoins, c'est assez lourd en termes de temps de calcul, et il existe une méthode bien plus rapide avec la fonction $InRange()$.

Tout d'abord, il faut définir un pixel "borne supérieure" et un pixel "borne inférieure" avec nos tolérances $\epsilon_1, \epsilon_2, \epsilon_3$ sur respectivement les composantes H,S et V. Nous veillerons aussi à restreindre les valeurs des bornes dans l'intervalle de définition d'une couleur en HSV, à savoir : $[[0,0,0];[179,255,255]]$. Une fois cela fait, on fait appel à la fonction $InRange()$ qui nous renvoie le masque de l'image.

Par dichotomie, j'ai fini par prendre $\epsilon_2 = 11$ et $\epsilon_2 = 110$. Le choix de $\epsilon_3 = 125$ permet d'exclure les zones beaucoup plus claires ou sombres que le pixel de référence. Dans la pratique, cela nous permet de limiter grandement le bruit.

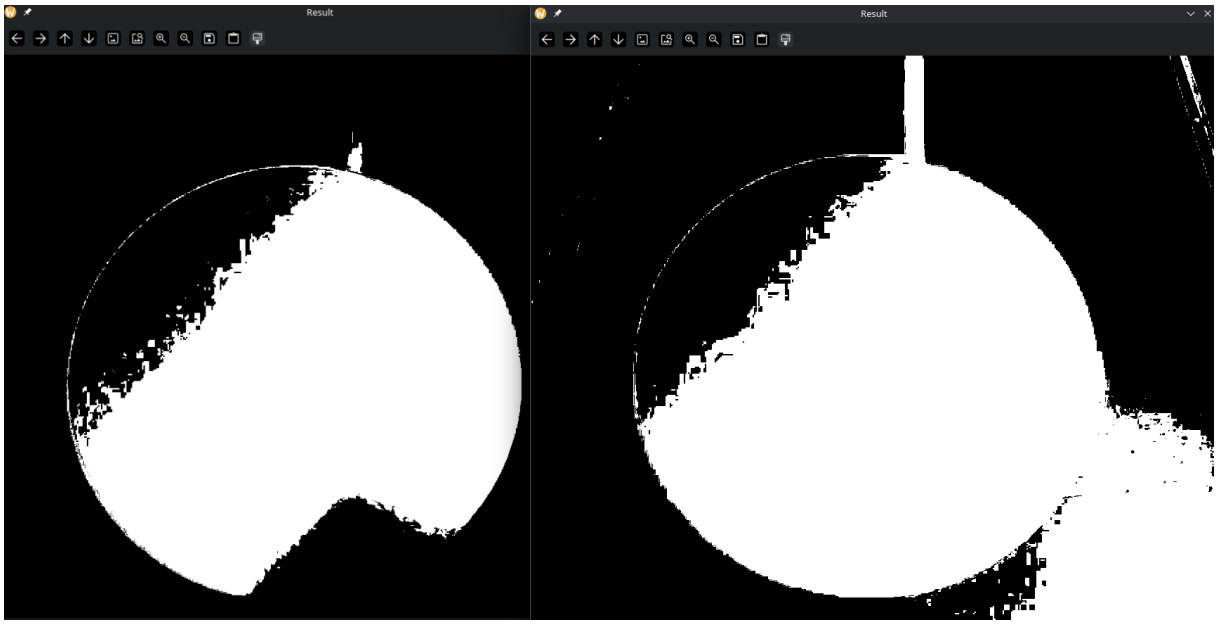


FIGURE I.1 – Comparaison entre $\epsilon_3 = 125$ (gauche) et $\epsilon_3 = 255$ (droite)

```
# On definit les bornes inferieure et superieure comprises entre [0,0,0]
    et [179,255,255] pour la detection de la couleur
lower_bound = np.clip(np.array([color[0]-epsilon, color[1]-epsilon2,
   color[2]-epsilon3]), [0,0,0], [179,255,255]).astype(np.uint8)
upper_bound = np.clip(np.array([color[0]+epsilon, color[1]+epsilon2,
   color[2]+epsilon3]), [0,0,0], [179,255,255]).astype(np.uint8)
imgresult = cv.inRange(imgHSV, lower_bound, upper_bound)
```

Code I.2 – Code du calcul du masque de l'image

I.3 Érosion, dilatation et gradient

Avant d'appliquer les opérations morphologiques (érosion, dilatation, gradient, etc.), nous devons définir un noyau qui servira de base pour ces opérations. On prendra une matrice remplie de 1 pour toutes les opérations. La taille de la matrice variera selon la puissance que nous souhaitons donner à l'opération. Un noyau plus grand amplifie les effets de l'opération, mais il augmente le temps de calcul. Cependant, des effets amplifiés ne sont pas toujours une bonne chose, car plus les effets sont importants, plus l'image résultante des ouvertures et des fermetures sera différente de l'image initiale. On peut donc perdre des informations utiles.

Initialement, je pensais que le mieux serait d'abord de faire une ouverture (érosion puis dilatation), ce qui permettrait d'éliminer les points isolés. Suivie d'une fermeture (dilatation puis érosion) qui permettrait de combler les éventuels "trous" à l'intérieur de la balle.

Dans la pratique, c'est bel et bien le cas si nous effectuons la recherche de la balle sur l'image entière, car il risque d'y avoir beaucoup de bruit. Nous verrons dans la partie II.4 que ce n'est pas nécessairement le meilleur choix si l'on souhaite obtenir la meilleure précision pour le suivi de la balle, à condition d'avoir préalablement réduit l'espace de travail.

Après avoir effectué nos éventuelles ouvertures et fermetures, nous allons appliquer un gradient afin d'extraire les contours de l'image.

```
# Erosion et dilatation (Ouverture)
kernel_ouverture = np.ones((3,3),np.uint8)
iterations_ouverture = 1
erosion1 = cv.erode(imgresult, kernel_ouverture, iterations_ouverture)
dilation1 = cv.dilate(erosion1, kernel_ouverture, iterations_ouverture)

# Dilatation et erosion (Fermeture)
kernel_fermeture = np.ones((5,5),np.uint8)
iterations_fermeture = 1
dilation2 = cv.dilate(dilation1, kernel_fermeture, iterations_fermeture)
erosion2 = cv.erode(dilation2, kernel_fermeture, iterations_fermeture)

# Gradient
kernel_grad = np.ones((3,3),np.uint8)
gradient = cv.morphologyEx(erosion2, cv.MORPH_GRADIENT, kernel_grad)
```

Code I.3 – Code des opérations morphologiques

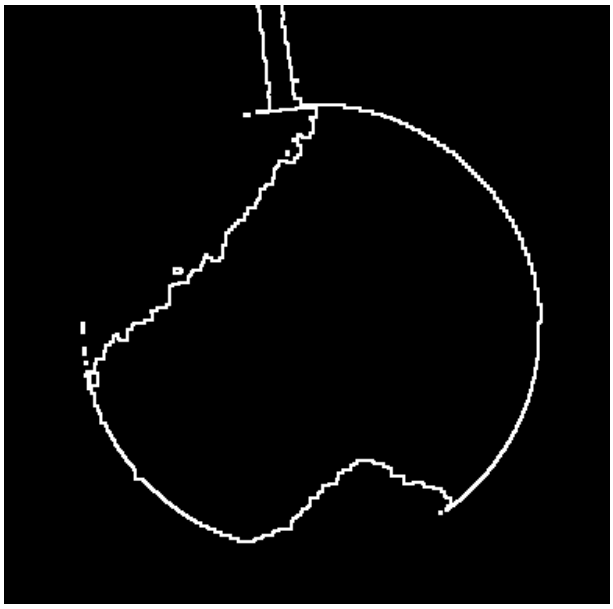


FIGURE I.2 – Exemple d'image post-gradient

I.4 Transformée de Hough et tracé du cercle sur l'image d'origine

Avec notre gradient, nous pouvons désormais appliquer la transformée de Hough, qui renvoie la liste des cercles potentiels définis par leur centre et leur rayon. Le cercle de rang 0 est celui qui a reçu le plus de votes. Concernant les valeurs des paramètres de la fonction, je les ai fixées de manière empirique.

```
# Transformée de Hough pour détecter les cercles
rows = gradient.shape[0]
circles = None
circles = cv.HoughCircles(
    gradient,           # Image d'entrée (en noir et blanc)
    cv.HOUGH_GRADIENT, # Méthode de détection
    1,
    rows / 8,           # minDist : Distance minimale entre les centres
        des cercles détectés
    param1=100,
    param2=20,
    minRadius=10,       # Rayon minimum des cercles à détecter
    maxRadius=500       # Rayon maximum des cercles à détecter
    )
```

Code I.4 – Code de la Transformée de Hough

Nous allons seulement retenir le cercle ayant le plus de votes et tracer son centre ainsi que son périmètre. Nous aborderons l'offset en partie II.3.

```
# Dessin du cercle détecté
output = img.copy()
if circles is not None:
    # Arrondi les valeurs des coordonnées et du rayon
    circles = np.uint16(np.around(circles))
    # On garde le cercle avec le plus de votes
    i=circles[0, 0]
    # Centre du cercle dans l'image réduite
    center = (i[0], i[1])
    # Centre du cercle dans l'image originale
    center_original_image = (i[0]+offset[0], i[1]+offset[1])
    radius = i[2]
    # Dessiner le centre du cercle
    cv.circle(output, center_original_image, 1, (0, 255, 0), 3)
    # Dessiner le contour du cercle
    cv.circle(output, center_original_image, radius, (255, 0, 255), 3)
```

Code I.5 – Code du tracé du cercle détecté

II – Identification de la balle sur une vidéo

II.1 Lecture de la vidéo image par image

Nous pouvons considérer une vidéo comme un tableau d'images. Idéalement, nous voudrions traiter l'image avec le même nombre de fps que la vidéo originale, mais pour des raisons de temps de calcul, cela ne va pas être possible. Nous allons donc, dans un premier temps, récupérer le fps de la vidéo originale et nous fixerons de notre côté le fps de notre traitement d'image. En d'autres termes, nous allons donc sauter des images de la vidéo originale afin de compenser le temps de calcul nécessaire pour le traitement de chaque image.

```
while cap.isOpened():

    start_time = time.time()

    if color_selected == True:
        # Passe à la prochaine frame
        cap.set(cv.CAP_PROP_POS_FRAMES, int(cap.get(cv.
            CAP_PROP_POS_FRAMES) + fps_original/fps))

    ret, frame = cap.read()
    if not ret:
        break

    cv.imshow('Frame', frame)
    if color_selected == False:
        cv.setMouseCallback('Frame', get_color_HSV, param=(frame,R))
        # Attends que l'utilisateur clique pour déclancher le callback
        while not color_selected : cv.waitKey(1)
        # Désactive le callback
        cv.setMouseCallback('Frame', lambda *args : None)
        start_time = time.time()

    if color_HSV is not None:
        color_HSV, center, radius = find_ball(frame, epsilon, epsilon2,
            color_HSV, center, radius)

    # Temps de calcul du traitement de la frame
    elapsed_ms = int((time.time() - start_time) * 1000)
    # Temps d'affichage d'une frame en theorie - temps de calcul
    wait_time = max(1,frame_duration_ms - elapsed_ms)
    #wait_time = 0 # Pour faire du pas à pas

```

```
    # Attends wait_time ms puis sort du while si 'q' pressé
    if cv.waitKey(wait_time) & 0xFF == ord('q'):
        break
```

Code II.1 – Code de la boucle de lecture de la vidéo sans rattrapage du retard de lecture

À chaque itération de la boucle while, nous allons attendre un certain temps avant d'afficher l'image suivante afin de contrôler la vitesse de lecture de la vidéo et de garantir une lecture fluide et identique à celle de la vidéo originale, même si nous affichons moins d'images. Nous souhaitons que le temps d'affichage de chaque image soit toujours de $fps_{original}/fps_{désiré}$. Afin de compenser le temps de calcul de notre traitement de chaque image, correspondant à la fonction $find\_ball()$, nous allons mesurer son temps d'exécution. Ensuite, nous retrancherons le temps d'exécution du temps d'attente souhaité. Le souci de cette méthode de lecture est que, si le temps de calcul est supérieur au temps d'affichage de l'image, alors on prend du retard dans la lecture de la vidéo, ce qui crée une impression de ralentissement.

Afin d'éviter ces ralentissements, nous pouvons mesurer le retard que nous avons obtenu sur chaque image et ensuite sauter dans la vidéo le nombre d'images correspondant à ce retard pour retrouver la vitesse de lecture de la vidéo originale.

```
while cap.isOpened():

    start_time = time.time()

    if color_selected == True:
        # Passe à la prochaine frame
        cap.set(cv.CAP_PROP_POS_FRAMES, int(cap.get(cv.
            CAP_PROP_POS_FRAMES) + fps_original/fps + time_skip*
            fps_original))
        # Le + time_skip*fps_original permet de rattraper le retard
            obtenu si le traitement de la frame est plus long que sa duré
            e d'affichage

    # Si il n'y a pas de prochaine frame, on sort du while
    ret, frame = cap.read()
    if not ret:
        break

    cv.imshow('Frame', frame)
    if color_selected == False:
        cv.setMouseCallback('Frame', get_color_HSV, param=(frame,R))
         # Attends que l'utilisateur clique pour déclancher le callback
        while not color_selected : cv.waitKey(1)
        # Désactive le callback
        cv.setMouseCallback('Frame', lambda *args : None)
        start_time = time.time()

    if color_HSV is not None:
        color_HSV, center, radius = find_ball(frame, epsilon, epsilon2,
            color_HSV, center, radius)

    time_skip = 0

    # Temps de calcul du traitement de la frame
    elapsed_ms = int((time.time() - start_time) * 1000)
```

```
    if elapsed_ms > frame_duration_ms:
        wait_time = 1
        # Temps à rattraper en secondes
        time_skip = (elapsed_ms - frame_duration_ms)/1000
        print("Skip time :", time_skip)
    else :
        # Temps d'affichage d'une frame en theorie - temps de calcul
        wait_time = max(1,frame_duration_ms - elapsed_ms)

    # Attends wait_time ms puis sort du while si 'q' pressé
    if cv.waitKey(wait_time) & 0xFF == ord('q'):
        break
```

Code II.2 – Code de la boucle de lecture de la vidéo avec rattrapage du retard de lecture

II.2 Mise à jour de la couleur à chaque image

Étant donné que la couleur de référence de la balle est extraite manuellement par l'utilisateur sur la première image, il peut être intéressant de mettre à jour notre couleur de référence tout au long de la lecture de la vidéo. Cela nous permettra d'obtenir de meilleurs résultats si, durant la vidéo, les composantes de couleur de la balle changent. Ces changements peuvent être dus, par exemple, à un déplacement de la caméra, à un nuage qui passe, etc.

Nous allons donc, après chaque détection de cercle sur une image, extraire un carré de côté $rayon/5$, centré sur le centre du cercle détecté. Ensuite, nous obtiendrons notre nouvelle couleur de référence à partir de la médiane de chacune des composantes HSV des pixels de la zone. Si aucun cercle n'a été trouvé par notre traitement de l'image actuelle, nous restaurerons la couleur à sa valeur originale. Cela permettra d'éviter que l'algorithme ne se perde. Par exemple, si, lors du traitement d'une image, on trouve un centre qui ne correspond absolument pas au centre de la balle, notre nouvelle couleur ne correspondra plus du tout à la couleur réelle de la balle. Donc, il nous sera impossible de détecter la balle lors des itérations suivantes. C'est pour cela qu'il est important de pouvoir restaurer la couleur initiale si nous ne détectons plus aucun cercle.

```
if circles is not None:
    R = radius//10
    # Define the region of interest (ROI)
    roi = imgHSV[center[1]-R:center[1]+R, center[0]-R:center[0]+R]
    new_color = np.median(roi, axis=(0, 1))
    print("New color (HSV): ", new_color)
else :
    # Si on n'a pas detecté de cercle, on restaure la couleur de la
       balle originale
    new_color = color_HSV_original.copy()
    print("No circle detected, back to original color")
```

Code II.3 – Code de la mise à jour de la couleur de référence

II.3 Réduction de l'espace de travail

En essayant de traiter la vidéo à pleine vitesse, nous nous rendons rapidement compte que la vitesse de calcul est le nerf de la guerre si nous souhaitons pouvoir lire la vidéo de manière fluide, avec un nombre de fps satisfaisant. Sachant que la complexité de nos fonctions de traitement d'image est de l'ordre de n^2 pour une image de $n \times n$ pixels. Si nous parvenons à réduire l'espace de travail de nos fonctions à une sous-image de l'image initiale qui comporte la balle, nous pourrions réduire significativement notre temps de calcul par image. Par exemple, en divisant n par 2, nous diviserions le temps de calcul par 4.

Supposons qu'entre deux images successives, la balle n'ait pas eu le temps de se déplacer beaucoup. Nous pouvons alors, pour chaque image, extraire un carré de coté $2 \times rayon_{n-1} \times marge$, centré sur le centre du cercle de l'image $n-1$, et supposer que la balle sera très probablement à l'intérieur. Nous penserons à calculer l'offset qu'il sera nécessaire d'ajouter aux coordonnées du centre du cercle trouvé dans l'image réduite afin de retrouver ses coordonnées dans l'image initiale. Si nous ne trouvons pas la balle à l'intérieur, pour l'image suivante, nous ne réduirons pas l'espace de travail. Idéalement, il faudrait reprendre le traitement de l'image actuelle dans son ensemble pour essayer de trouver le cercle. Sauf que, même si nous finissons par le trouver, notre temps de calcul pour cette image aura été trop élevé et nous risquons de prendre du retard dans la lecture de la vidéo. C'est pour cette raison que nous nous contenterons de passer à l'image suivante.

```
1  if old_center is not None and old_radius is not None:
2      img_short = img[max(0,int(old_center[1]-old_radius*(1+margin)))
3                      :min(img.shape[0],int(old_center[1]+old_radius*(1+
                           margin))),
4                      max(0,int(old_center[0]-old_radius*(1+margin)))
5                      :min(img.shape[1],int(old_center[0]+old_radius*(1+
                           margin)))]
6      # Pour obtenir les coordonnées du cercle détecté (x,y) dans l'image
           originale
7      offset = (max(0,int(old_center[0]-old_radius*(1+margin))),
8                max(0 ,int(old_center[1]-old_radius*(1+margin))))
9  else:
10     img_short = img
11     offset = (0,0)
```

Code II.4 – Code de réduction de l'espace de travail

II.4 Amélioration de la précision de l'identification de la balle

J'ai testé différentes combinaisons d'ouvertures et fermetures, avec un noyau différent. Puis j'ai été surpris de constater que la détection de la balle était la plus précise lorsque je ne faisais aucune ouverture. Initialement, je me doutais que ces opérations pouvaient entraîner une perte d'information, mais je supposais que c'était négligeable. Afin de comprendre ce qu'il se passait réellement, j'ai comparé mon image avant et après ouverture pour les images sur lesquelles la détection perdait en précision.

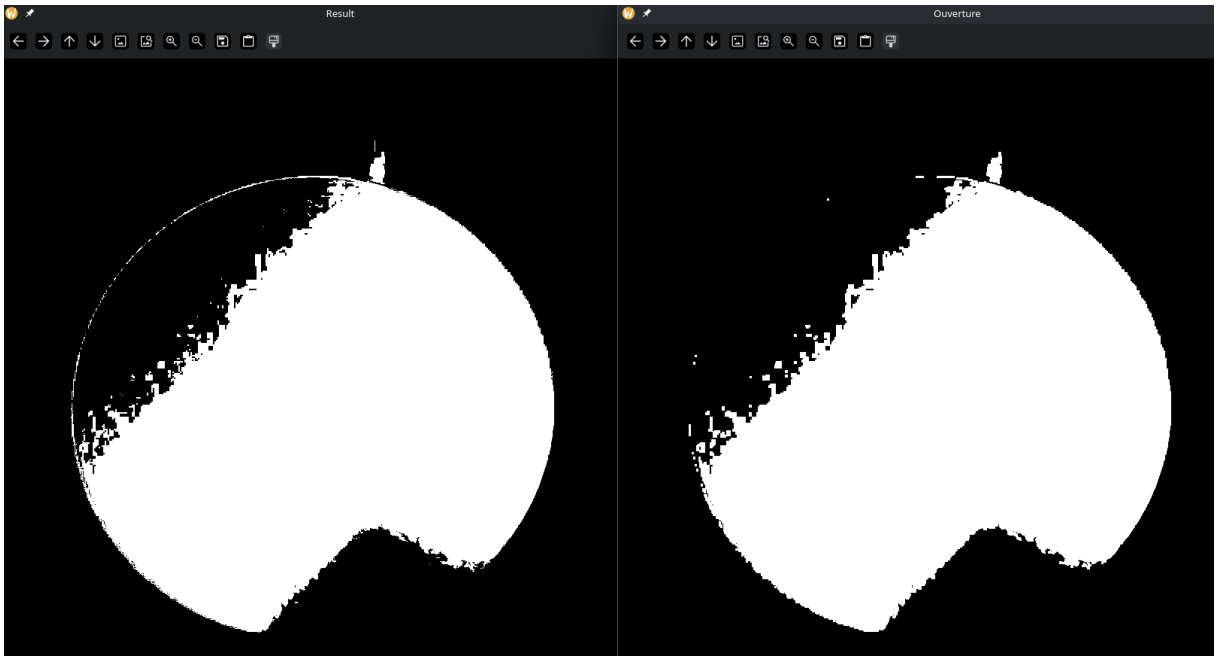


FIGURE II.1 – Comparaison avant ouverture (gauche) et après ouverture (droite)

Sur cette image, nous ne parvenons pas à sélectionner les pixels en haut à gauche de la balle, car ils sont trop lumineux. Je ne sais pas vraiment expliquer pourquoi, mais nous parvenons à sélectionner les pixels du contour de la balle. Nous pouvons voir que l'arc de cercle en haut à gauche de la balle est supprimé par l'érosion de l'ouverture, car il est trop fin. Néanmoins, même s'il est fin, il apporte des informations très utiles si l'on souhaite intégrer un cercle dans l'image. Ainsi, ne pas effectuer d'ouverture sur l'image nous permet d'améliorer notre précision. Cela parait assez contre intuitif, car en temps normal, les ouvertures permettent d'éliminer les pixels de bruit et donc, en général, d'améliorer la précision. Dans notre cas, le fait de commencer par une ouverture (donc une érosion) supprime l'arc. Et en ce qui concerne les pixels de bruit, nous en avons très peu, car nous avons préalablement réduit l'espace de travail, ce qui limite grandement la présence de pixels de bruit. Cela explique pourquoi nous obtenons de meilleurs résultats avec seulement une fermeture.

Concernant les temps de calcul, ne pas faire d'ouverture va rallonger les temps de calcul, car nous aurons des points isolés qui resteront apparents sur le gradient. Cela signifie qu'il y aura d'avantage de points à considérer pour la transformée de Hough et donc un temps de calcul plus long. En revanche, la fermeture va combler les trous isolés et donc simplifier le gradient, réduisant ainsi le temps de calcul de la transformée de Hough.

En conclusion, si l'on souhaite avoir la meilleure précision possible, quitte à rallonger les temps de calcul, il faut effectuer seulement une fermeture. Mais si l'on accorde plus d'importance à la vitesse du traitement et donc à la quantité d'images traitées par seconde, quitte à perdre un peu en précision, il faut alors effectuer une ouverture, puis une fermeture.