Suivi d'un objet coloré dans une vidéo

1. Introduction

On souhaite suivre le déplacement d'un objet dans une vidéo. On suppose que l'objet est identifiable par sa couleur. La première vidéo étudiée, prise à 60 images par secondes, montre une bille bleue entrant en collision avec une bille rouge (sur un plan horizontal). Il s'agit de mettre au point un algorithme capable de suivre le centre de chacune des deux billes, sans risque de confusion au moment où les billes se touchent.

On fera appel aux bibliothèques `opencv`, `numpy`, `matplotlib`.

```
import cv2 as cv
import numpy
import matplotlib.pyplot as plt
```

2. Segmentation par couleur

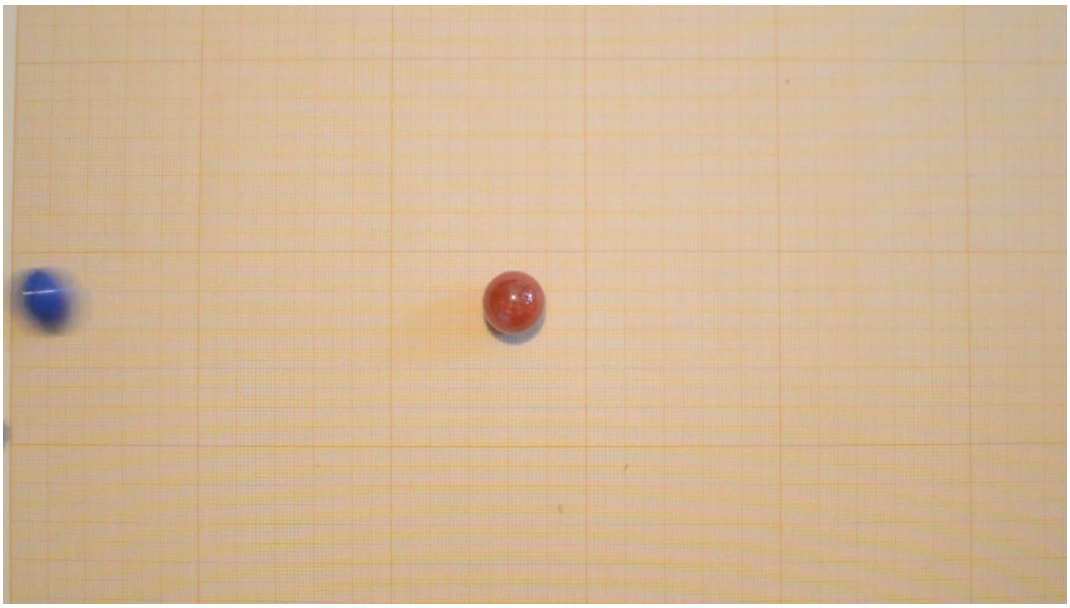
2.a. Représention HSV des couleurs

La première étape consiste à extraire les pixels de l'objet en utilisant l'information de couleur. Le but est d'obtenir une image dans laquelle les pixels de l'objet sont blancs (valeur 255) et les autres pixels sont noirs (valeur 0). On dit alors que l'image est *segmentée*.

Voici comment se fait l'ouverture du fichier vidéo et la lecture jusqu'à la première image qui contient les deux billes :

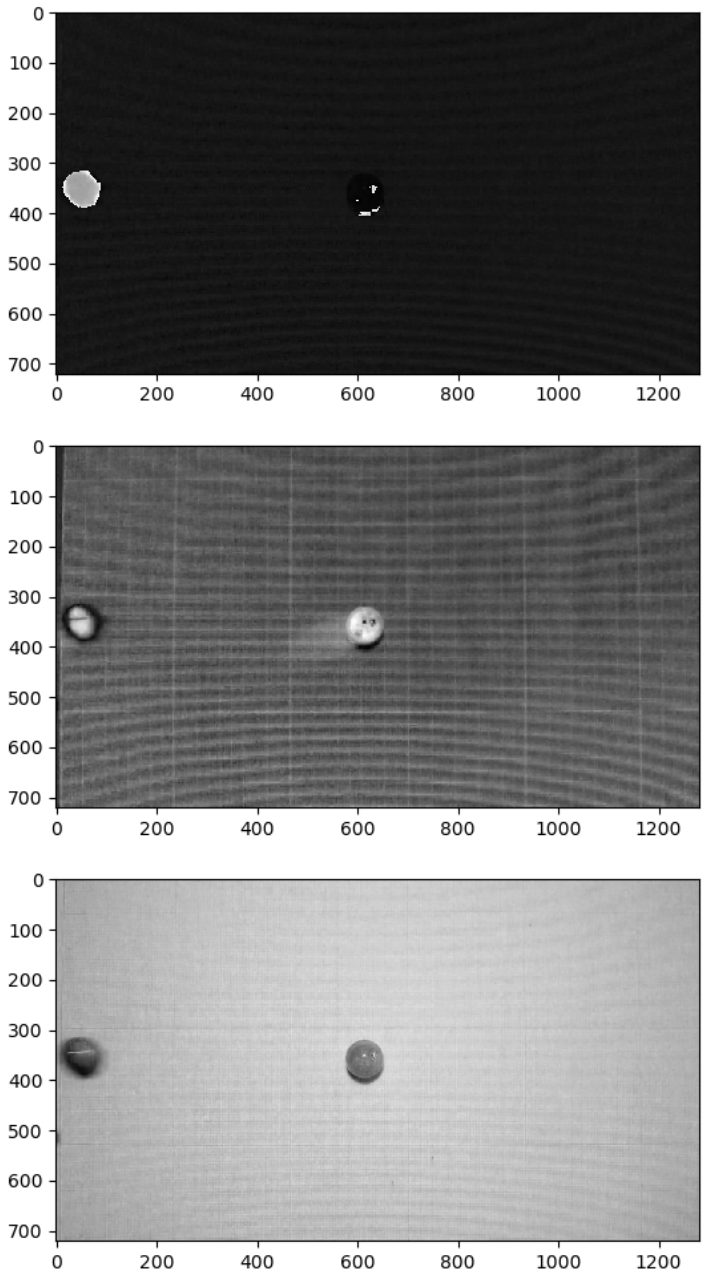
```
video = cv.VideoCapture("billes-2.mp4")

def onMouse(event,x,y,flags,param):
    global i0,j0
    if event==cv.EVENT_LBUTTONUP:
        j0=x
        i0=y
        print(i0,j0)

cv.namedWindow("frame")
cv.setMouseCallback('frame',onMouse)
f=0
while f<141:
    success,frame=video.read()
    f += 1
cv.imshow('frame',frame)
cv.waitKey()
```

L'image est affichée dans une fenêtre. La fonction `onMouse` permet d'afficher les indices du pixel pointé lorsqu'on clique avec le bouton de la souris. Cela permettra de faire des repérages sur l'image.

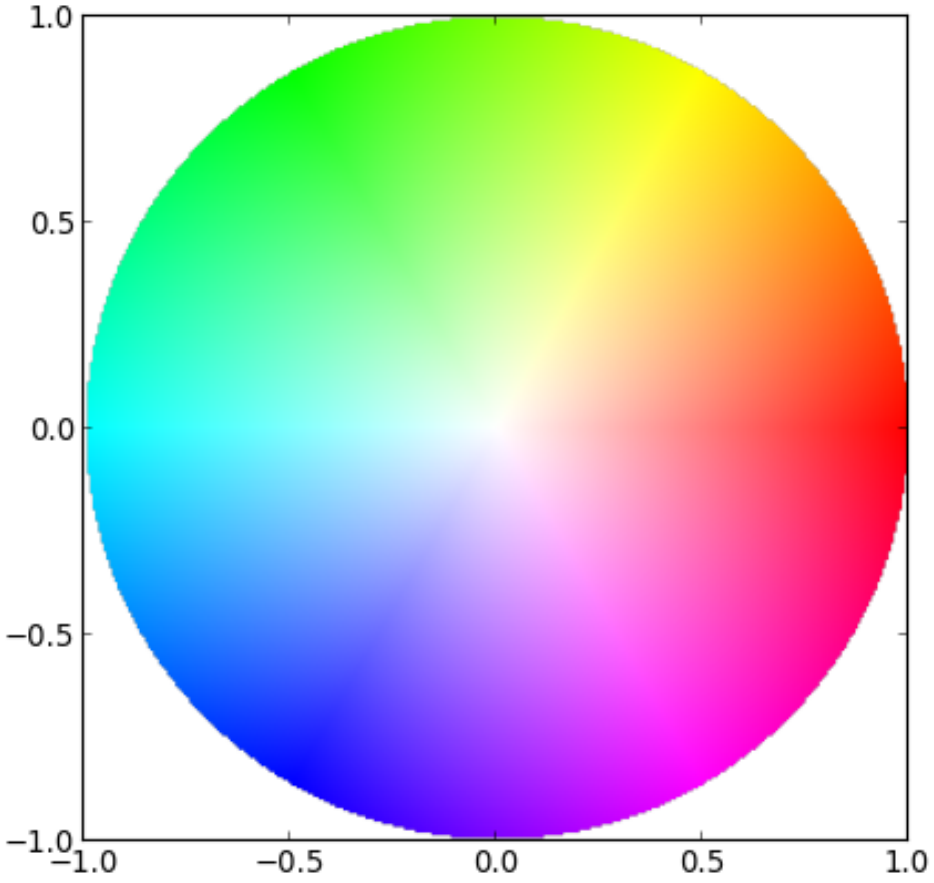


L'image comporte trois couches Bleue Verte Rouge (dans cet ordre). Pour le traitement d'image, il est préférable d'utiliser la représentation des couleurs HSV (Hue : teinte, Sat : saturation, Val : valeur). Voici comment se fait la conversion avec opencv et l'affichage des couches HSV avec matplotlib :

```
hsv = cv.cvtColor(img,cv.COLOR_BGR2HSV)
hue = hsv[:,:,0]
sat = hsv[:,:,1]
val = hsv[:,:,2]
plt.figure(figsize=(8,12))
plt.subplot(311)
plt.imshow(hue,cmap=plt.cm.gray)
plt.subplot(312)
plt.imshow(sat,cmap=plt.cm.gray)
plt.subplot(313)
plt.imshow(val,cmap=plt.cm.gray)
```



La couche H représente la teinte, une valeur comprise entre 0 et 180. La teinte est parfois représentée sur un cercle, avec une valeur d'angle en degré double de celle-ci :



Les couches S (saturation) et V (valeur ou luminance) comportent des valeurs comprises entre 0 et 255. Nous constatons que la bille rouge située au centre a une teinte très proche du fond. Il est donc impossible de discerner cet objet en utilisant seulement l'information de teinte. En revanche, sa saturation est beaucoup plus élevée. On remarque aussi la saturation plus élevée des traits du papier millimétré. La couche V montre que les deux billes ont une luminance beaucoup plus faible que le fond. Il serait donc possible d'effectuer une extraction des pixels d'une bille en utilisant la couche V. Cependant, nous voulons une méthode capable de discerner les deux billes, ce qui est indispensable lorsque les deux billes se touchent. Nous devons donc utiliser les informations des couches H et S. Par ailleurs, une méthode reposant sur la couleur permettra de chercher un objet d'une couleur donnée sans savoir *a priori* sa position dans l'image.

2.b. Histogrammes HS

On commence par isoler un rectangle inscrit dans la bille rouge. Les indices des coins opposés du rectangle sont obtenus en cliquant à la souris. Voici l'extraction du rectangle :

```
region = frame[334:384,594:638,:]
cv.imshow('frame',region)
cv.waitKey()
```

```
hsv = cv.cvtColor(region,cv.COLOR_BGR2HSV)
```

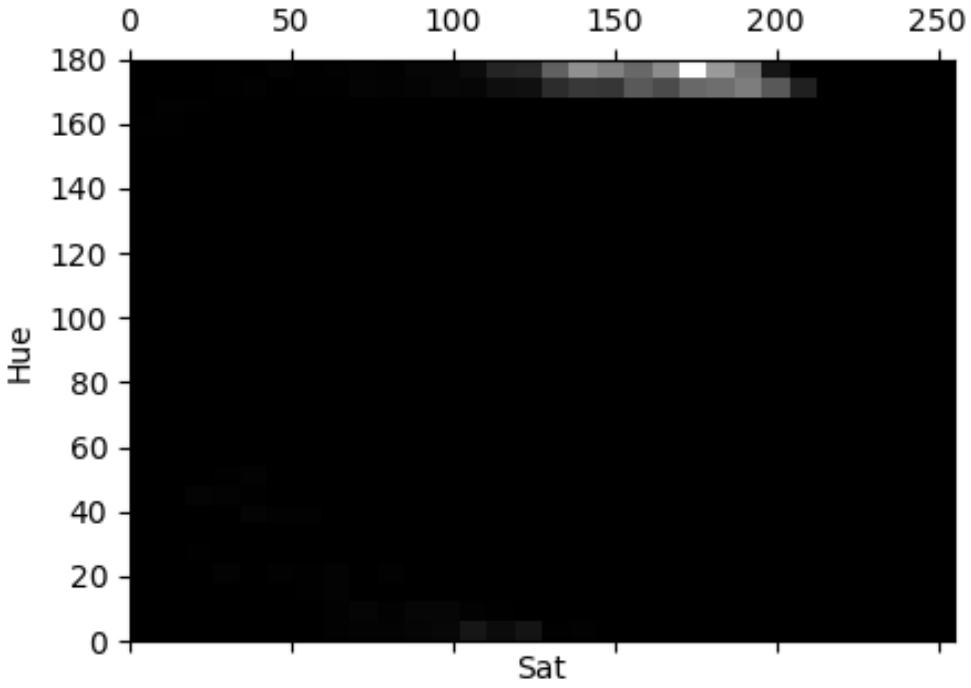
L'image extraite est affichée pour vérification, puis convertie en HSV.

Nous devons réaliser un histogramme HS de cette image. Il s'agit d'un histogramme 2D, c'est-à-dire une matrice dont chaque élément représente un rectangle dans le plan HS, contenant le nombre de pixels de l'image dont les valeurs de H et S sont dans ce rectangle. Pour construire un histogramme, il faut diviser l'intervalle des valeurs de H ([0,180]) et celui des valeurs de S ([0,255]). Voici le calcul d'un histogramme comportant 30 subdivisions sur chaque axe :

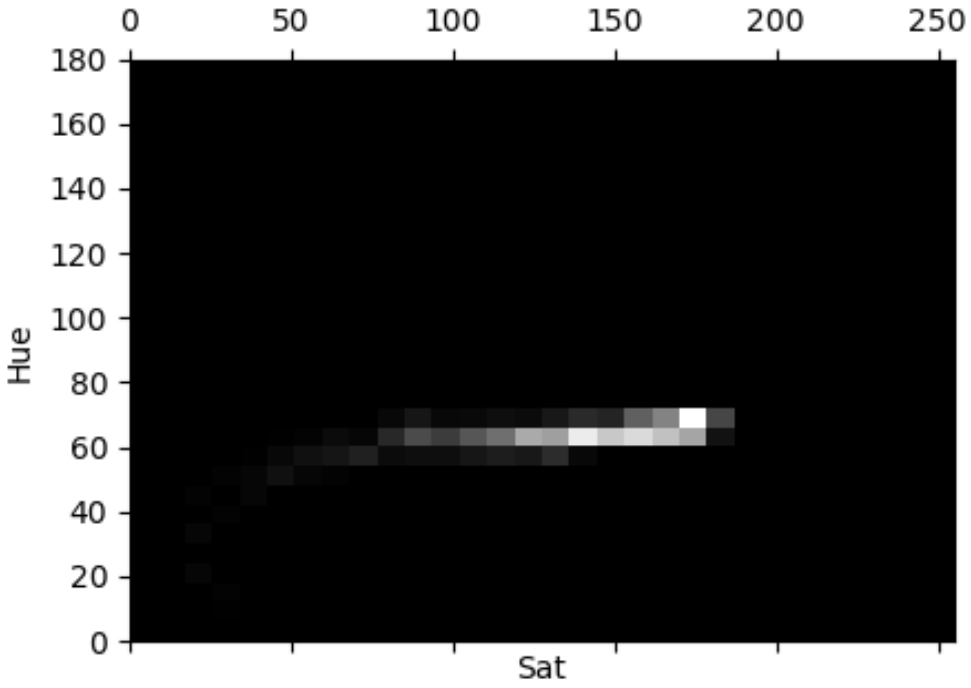
```
hist1 = cv.calcHist([hsv],[0,1],None,[30,30],[0,180,0,255])
cv.normalize(hist1, hist1, alpha=0, beta=255, norm_type=cv.NORM_MINMAX)
```

Le premier argument fournit les images utilisées pour le calcul de l'histogramme (ici une seule). Le second argument indique les couches utilisées, ici les couches 0 et 1. La normalisation permet d'obtenir un histogramme dont les valeurs sont comprises entre 0 et 255. Voici une représentation de l'histogramme sous la forme d'une image en niveaux de gris :

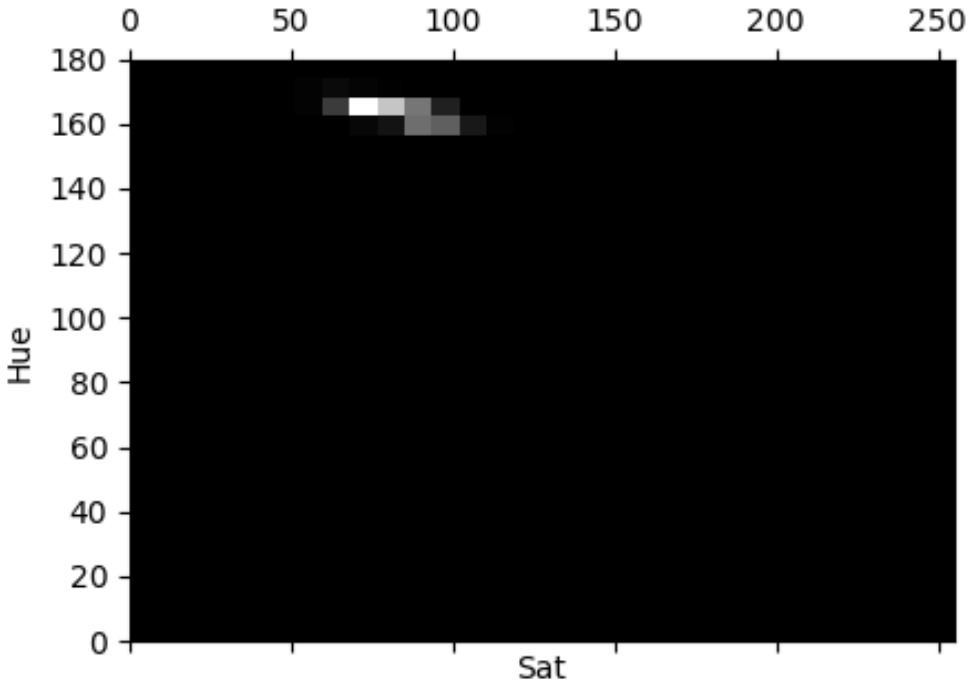
```
plt.matshow(hist1,extent=[0,180,0,255],cmap=plt.cm.gray)
plt.xlabel("Hue")
plt.ylabel("Sat")
```



On constate que la teinte prépondérante a une valeur d'environ 175, avec un niveau de saturation élevé. Voici l'histogramme de la bille bleue, obtenu de manière similaire :



La teinte dominante a une valeur d'environ 70 et la saturation s'étend sur un intervalle très large. Dans le plan HS, les zones non nulles des histogrammes des deux billes sont bien distinctes, ce qui montre qu'elles sont bien discernables en utilisant les couches H et S. Nous devons aussi vérifier qu'elles sont discernables du fond. Voici l'histogramme du fond :

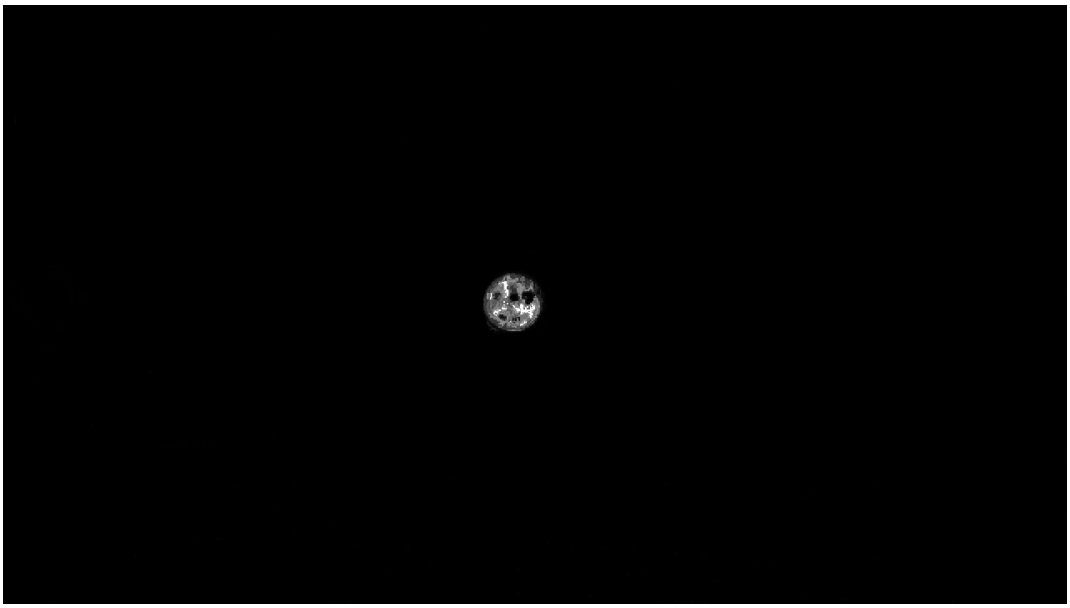


Comme déjà observé, le fond a une teinte voisine de la bille rouge mais son niveau de saturation beaucoup plus faible permet de le discerner.

2.c. Rétroprojection d'histogramme

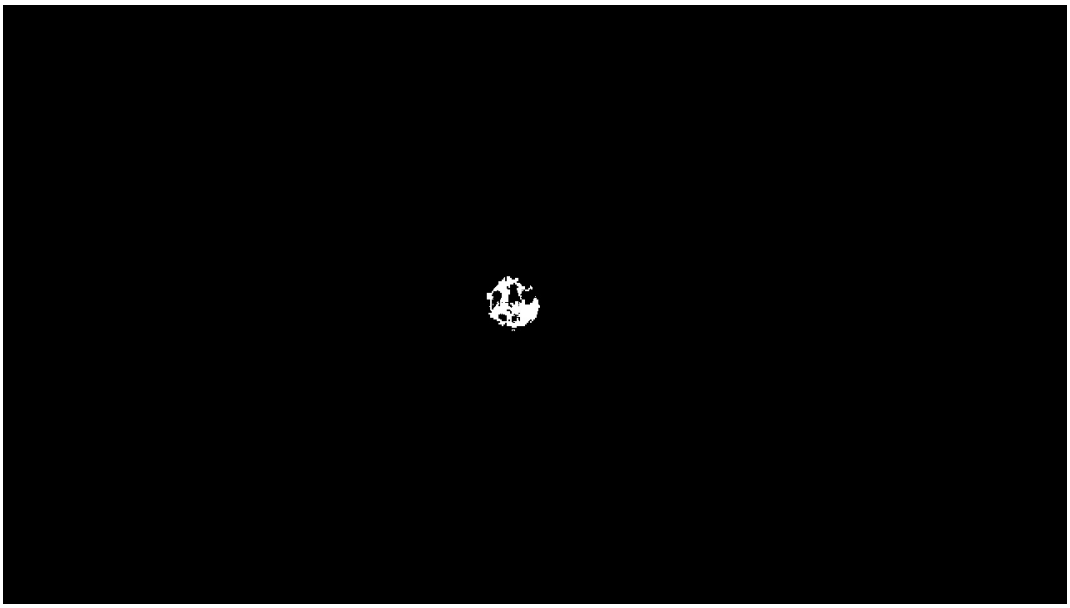
Pour extraire les pixels de chaque objet, nous allons utiliser la technique de *rétroprojection d'histogramme*. Pour extraire la bille rouge, on utilise son histogramme `hist1`, c'est-à-dire un histogramme construit à partir d'un rectangle inscrit dans la bille. La rétroprojection consiste à construire une image en niveaux de gris, en remplaçant chaque pixel par la valeur de l'histogramme correspondant aux valeurs H et S de ce pixel. Voici comment se fait la rétroprojection :

```
back_proj = cv.calcBackProject([hsv],[0,1],hist1,[0,180,0,255],scale=1)
```



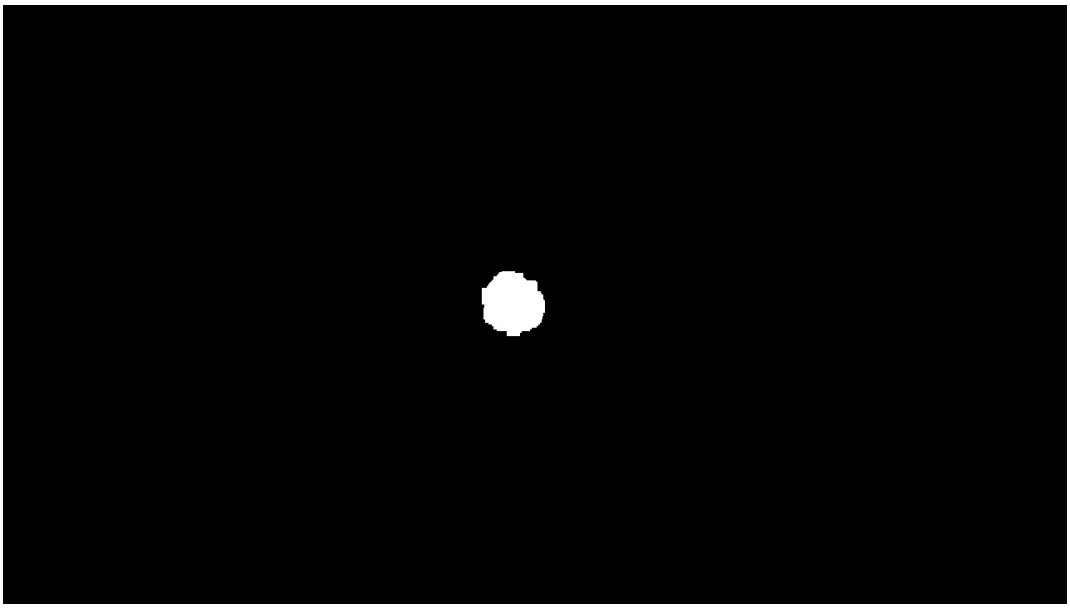
La rétroprojection donne une valeur non nulle seulement aux pixels dont la teinte et la saturation sont dans le domaine du plan HS où l'histogramme de la bille rouge présente des valeurs non nulles. L'étape suivante consiste à binariser l'image en appliquant un seuil :

```
r,img = cv.threshold(back_proj,100,255,cv.THRESH_BINARY)
```



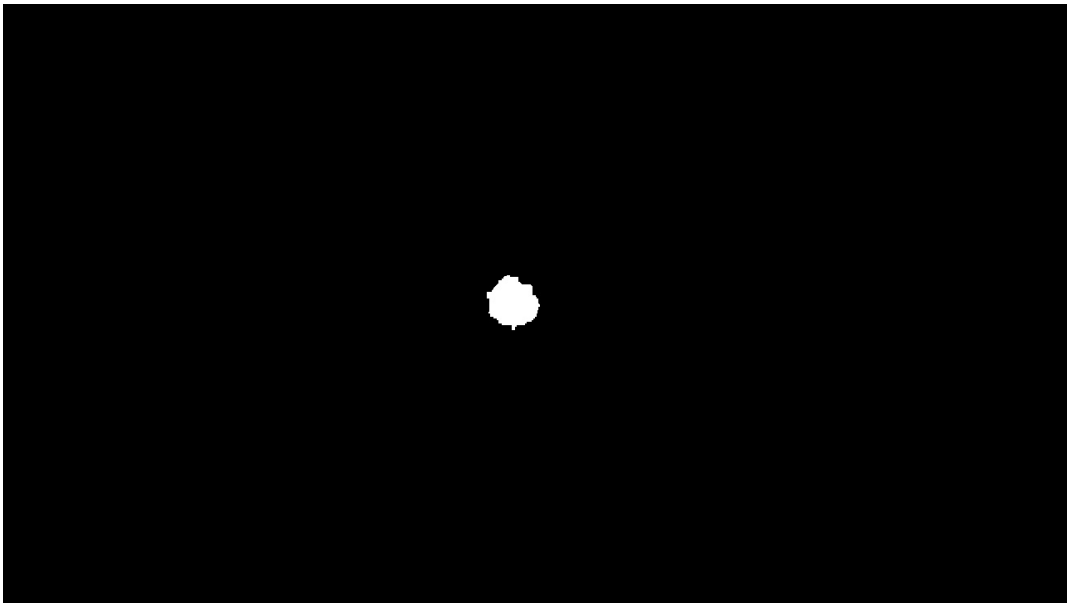
Les pixels blancs appartiennent à la bille mais il reste des zones noires à l'intérieur du disque. Nous souhaitons obtenir une représentation de l'objet qui corresponde au mieux à sa surface apparente, avec un ensemble de pixels connexes. Cela peut être obtenu avec une opération de *dilatation*. La dilatation d'une image binaire consiste à remplacer chaque pixel par la valeur maximale des pixels situés dans un voisinage carré centré sur ce pixel. Il s'agit d'une opération de filtrage non linéaire, par opposition au filtrage par convolution qui est linéaire. Si ce voisinage contient un pixel blanc, le pixel central devient blanc. Voici comment réaliser une dilatation en utilisant un voisinage carré de 13 par 13 pixels :

```
element = cv.getStructuringElement(cv.MORPH_RECT,(13,13))
img = cv.dilate(img,element)
```



La dilatation permet de supprimer les trous noirs mais elle a l'inconvénient d'élargir la zone. Dans le cas présent cela n'est pas gênant car on s'intéressera seulement au barycentre des pixels blancs. Pour réduire la zone, il suffit d'appliquer une érosion, qui consiste à remplacer chaque pixel par la valeur minimale du voisinage :

```
img = cv.erode(img,element)
```



Cette étape d'érosion n'est valable que si les trous noirs sont bien remplis par la dilatation.

3. Suivi de l'objet

3.a. Principe

Un objet extrait par segmentation se présente sous la forme d'un ensemble de pixels connexes que nous devons suivre au cours de son mouvement. Soient `(i0,j0)` les indices d'un pixel de l'image, en général à proximité de l'objet. On doit faire une recherche d'un pixel blanc en procédant par voisinage de taille croissante. La fonction suivante effectue cette recherche. Le voisinage est un carré de côté $2q+1$, où q est un entier augmenté itérativement tant qu'aucun pixel blanc n'est trouvé. La fonction renvoie les indices du pixel blanc trouvé.

```
def recherche_voisinage(A,M,N,i0,j0):
    if A[i0,j0]==255:
        return([i0,j0])
    q=0
    while True:
        q+=1
        for i in range(i0-q,i0+q+1):
            j=j0-q
            if i>=0 and i<M and j>=0 and j<N and A[i,j]==255:
                return([i,j])
            j=j0+q
            if i>=0 and i<M and j>=0 and j<N and A[i,j]==255:
                return([i,j])
        for j in range(j0-q+1,j0+q):
            i=i0-q
            if i>=0 and i<M and j>=0 and j<N and A[i,j]==255:
                return([i,j])
            i=i0+q
            if i>=0 and i<M and j>=0 and j<N and A[i,j]==255:
                return([i,j])
```

Une fois qu'un pixel appartenant à l'objet est trouvé, on doit construire la liste des pixels de l'objet, c'est-à-dire la liste des pixels blancs. Une méthode efficace pour faire cela consiste à colorier l'objet par diffusion. La méthode est récursive. Un pixel blanc est colorié en gris (on lui attribue la valeur 100) et enregistré dans la liste puis ces 4 voisins (horizontalement et verticalement) sont testés. La fonction est appelée récursivement sur chaque voisin blanc. L'ensemble des pixels de l'objet est ainsi balayé, à condition qu'ils soient connexes.

```
import sys
sys.setrecursionlimit(10000)

def test_pixel_objet(A,M,N,i,j,liste_pixels):
    liste_pixels.append([i,j])
    A[i,j]=100
    voisins=[(i+1,j),(i-1,j),(i,j-1),(i,j+1)]
    for (k,l) in voisins:
        if k>=0 and k<M and l>=0 and l<N:
            if A[k,l]==255:
                test_pixel_objet(A,M,N,k,l,liste_pixels)
```

Le barycentre des pixels de l'objet est calculé à partir de la liste des pixels par la fonction suivante :

```
def barycentre(liste_pixels):
    N=len(liste_pixels)
    xG=0.0
    yG=0.0
    for pixel in liste_pixels:
        xG += pixel[1]
        yG += pixel[0]
    xG /= N
    yG /= N
    return ([xG,yG])
```

On remarque que le premier indice du pixel correspond à l'ordonnée.

4. Boucle de traitement

La boucle de traitement lit les images de la vidéo et leur applique les opérations suivantes :

- ▷ Conversion au format HSV.
- ▷ Rétroprojection de l'histogramme HS de l'objet 1.
- ▷ Seuillage de l'image obtenue puis dilatation.
- ▷ Recherche d'un pixel de l'objet avec `recherche_voisinage`, à partir d'un pixel initial `(i1,j1)`.
- ▷ Obtention de la liste des pixels de l'objet avec `test_pixel_objet`.
- ▷ Calcul du barycentre des pixels avec `barycentre` et mémorisation dans une liste.
- ▷ Actualisation des indices `(i1,j1)` avec le barycentre.

La dernière opération place le pixel `(i1,j1)` à proximité de l'objet sur l'image suivante et même dans l'objet lorsque le déplacement de l'objet n'est pas trop grand. Toutes ces étapes doivent bien sûr être accomplies aussi pour l'objet 2, à partir de son histogramme.

Le code comportant la boucle de traitement est présenté ci-dessous. Une vidéo sur laquelle les barycentres sont marqués par une croix est aussi générée. Si `trace=True`, l'image de chaque objet après coloriage en gris est aussi affichée, ce qui permet de répérer d'éventuelles zones non coloriées, qui restent blanches. Un appui sur la touche q permet de stopper l'analyse.

La mise au point d'un algorithme de traitement d'image nécessite l'ajustement de certains paramètres. Dans le cas présent, ces paramètres sont :

- ▷ La taille des histogrammes HS.
- ▷ Le seuil de binarisation.
- ▷ La taille du noyau de la dilatation.

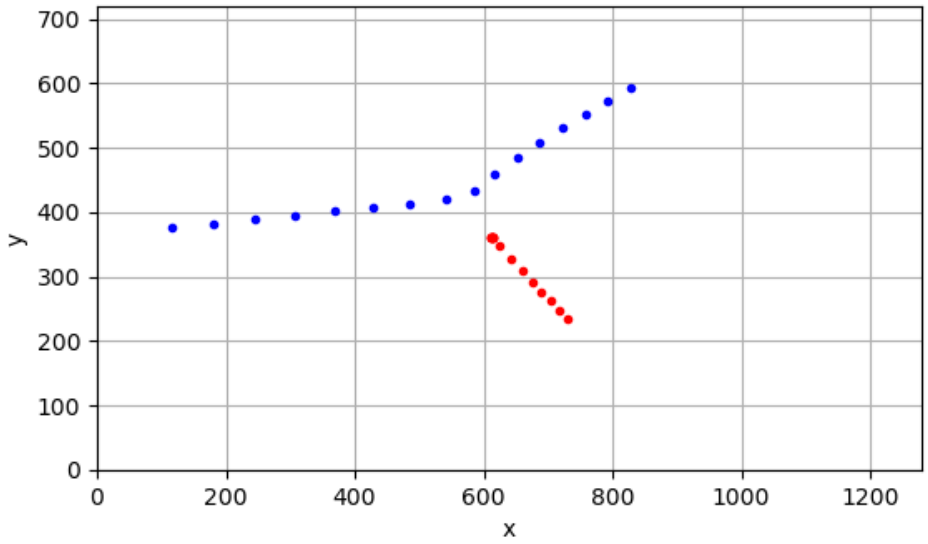
```
i1=364
j1=615
i2=357
j2=53
(M,N,c)=frame.shape
liste_G2 = []
liste_G1 = []
f=0
```

```
trace=True
videoWriter = cv.VideoWriter("billes-2-positions.avi",cv.VideoWriter_fourcc('I','4','2',

while success:
    success,frame=video.read()
    hsv = cv.cvtColor(frame,cv.COLOR_BGR2HSV)

    back_proj = cv.calcBackProject([hsv],[0,1],hist1,[0,180,0,255],scale=1)
    r,img = cv.threshold(back_proj,100,255,cv.THRESH_BINARY)
    img = cv.dilate(img,element)
    [i1,j1] = recherche_voisinage(img,M,N,i1,j1)
    liste_pixels=[]
    test_pixel_objet(img,M,N,i1,j1,liste_pixels)
    [xG,yG]=barycentre(liste_pixels)
    j1=int(xG)
    i1=int(yG)
    liste_G1.append([xG,M-yG])
    L=50
    img[i1-L:i1+L,j1]=200
    img[i1,j1-L:j1+L]=200
    if trace:
        cv.imshow('frame',img)
        if cv.waitKey(500)==113: break

    back_proj = cv.calcBackProject([hsv],[0,1],hist2,[0,180,0,255],scale=1)
    r,img = cv.threshold(back_proj,100,255,cv.THRESH_BINARY)
    img = cv.dilate(img,element)
    [i2,j2] = recherche_voisinage(img,M,N,i2,j2)
    liste_pixels=[]
    test_pixel_objet(img,M,N,i2,j2,liste_pixels)
    [xG,yG]=barycentre(liste_pixels)
    j2=int(xG)
    i2=int(yG)
    liste_G2.append([xG,M-yG])
    L=50
    img[i2-L:i2+L,j2]=200
    img[i2,j2-L:j2+L]=200
    if trace:
        cv.imshow('frame',img)
        if cv.waitKey(500)==113: break

    frame[i1-L:i1+L,j1]=[255,255,255]
    frame[i1,j1-L:j1+L]=[255,255,255]
    frame[i2-L:i2+L,j2]=[255,255,255]
    frame[i2,j2-L:j2+L]=[255,255,255]
    cv.imshow('frame',frame)
    videoWriter.write(frame)
    if cv.waitKey(500)==113: break
    f+=1

videoWriter.release()
```

4.a. Résultat

```
liste_G1=numpy.array(liste_G1)
liste_G2=numpy.array(liste_G2)
```

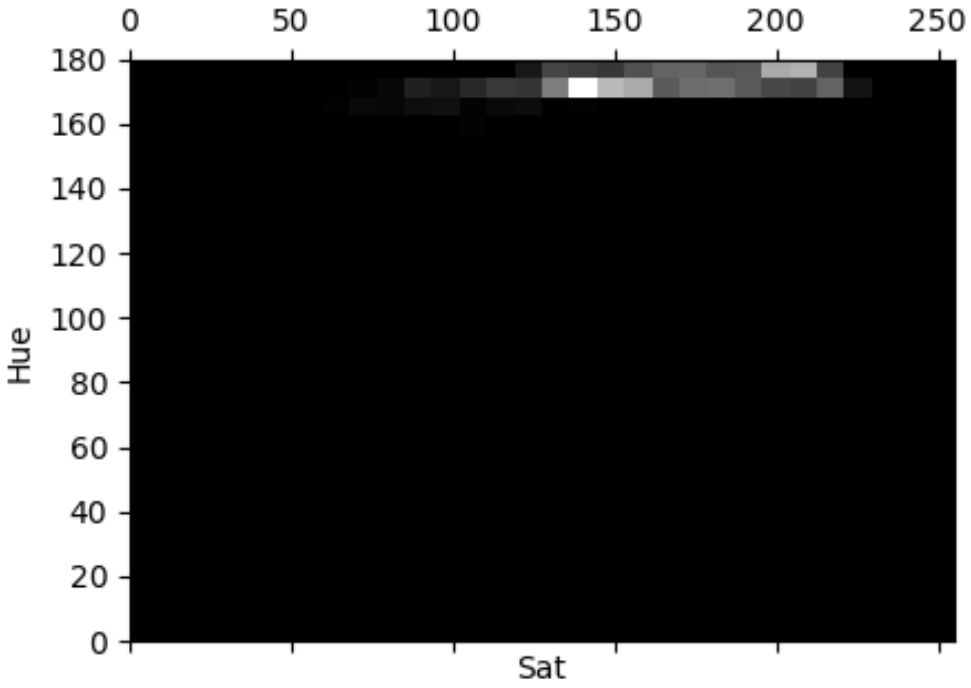
```
plt.figure()
plt.axes().set_aspect('equal')
plt.plot(liste_G1[:,0],liste_G1[:,1],"r.")
plt.plot(liste_G2[:,0],liste_G2[:,1],"b.")
plt.xlim(0,N)
plt.ylim(0,M)
plt.xlabel("x")
plt.ylabel("y")
plt.grid()
```



5. Suivi de l'enveloppe d'un objet

Considérons la vidéo ci-dessous, montrant le mouvement d'un objet dont la couleur n'est pas uniforme :

On constuit un histogramme HS à partir d'un rectangle sélectionné sur la partie orange de l'objet :



La teinte est très proche de celle du fond mais la saturation plus élevée permettra de discerner l'objet.

Voici le résultat d'une rétroprojection de cet histogramme sur l'image entière, suivi d'un seuillage :

```
hsv = cv.cvtColor(frame,cv.COLOR_BGR2HSV)
back_proj = cv.calcBackProject([hsv],[0,1],hist1,[0,180,0,255],scale=1)
r,img = cv.threshold(back_proj,100,255,cv.THRESH_BINARY)
```

Dans ce cas, la présence de zones noires de grande étendue nous a amené à effectuer une dilatation avec un noyau de très grande taille (31 par 31). La dilatation est suivie d'une érosion, de manière à réduire la taille de la zone.

```
element = cv.getStructuringElement(cv.MORPH_RECT,(31,31,))
img = cv.dilate(img,element)
img = cv.erode(img,element)
```

Le nombre de pixels à explorer dans l'objet est très grand et l'implémentation récursive de la fonction `test_pixel_objet` dépasse les capacités d'appels récursifs de Python. Nous préférons donc implémenter la recherche au moyen d'une pile :

```
def extraction_pixels_objet(A,M,N,i,j):
    liste_pixels = []
```

```
    pile = [[i,j]]
    while len(pile)!=0:
        [i,j] = pile.pop()
        liste_pixels.append([i,j])
        A[i,j]=100
        voisins=[(i+1,j),(i-1,j),(i,j-1),(i,j+1)]
        for (k,l) in voisins:
            if k>=0 and k<M and l>=0 and l<N:
                if A[k,l]==255:
                    pile.append([k,l])
    return liste_pixels
```

On se fixe pour objectif de déterminer l'enveloppe de l'objet. Au lieu de rechercher tous ses pixels, on peut donc rechercher seulement les pixels du bord de l'objet, c'est-à-dire les pixels dont au moins l'un des quatre voisins est noir :

```
def extraction_pixels_bord_objet(A,M,N,i,j):
    liste_pixels = []
    pile = [[i,j]]
    while len(pile)!=0:
        [i,j] = pile.pop()
        A[i,j]=100
        voisins=[(i+1,j),(i-1,j),(i,j-1),(i,j+1)]
        bord = False
        for (k,l) in voisins:
            if k>=0 and k<M and l>=0 and l<N:
                if A[k,l]==255:
                    pile.append([k,l])
                elif A[k,l]==0:
                    bord=True
            else:
                bord = True
        if bord :
            liste_pixels.append([i,j])
    return liste_pixels
```

La position de l'objet est déterminée comme le centre de son enveloppe rectangulaire de la manière suivante :

```
A = ((liste_pixels[:,1]).min(),(liste_pixels[:,0]).min())
B = ((liste_pixels[:,1]).max(),(liste_pixels[:,0]).max())
[xG,yG]=[(A[0]+B[0])/2,(A[1]+B[1])/2]
```

Plus généralement, il peut être intéressant de déterminer l'enveloppe convexe d'un ensemble de points, c'est-à-dire le plus petit polygone convexe contenant tous les points de l'ensemble. L'algorithme de construction de ce polygone est expliqué dans Enveloppe convexe. La liste des points (ici les points du bord de l'objet) doit être ordonnée par valeurs de i croissantes et par valeurs de j décroissantes. Pour faire ce tri, nous utilisons l'algorithme de tri par insertion. Voici tout d'abord une fonction triant une liste par valeurs de i croissantes :

```
def tri_insertion_x(liste_points):
    N = len(liste_points)
    for i in range(1,N):
        cle = liste_points[i]
        j = i-1
        while j>=0 and liste_points[j][0] > cle[0]:
            liste_points[j+1] = liste_points[j]
            j = j-1
        liste_points[j+1] = cle
```

Les points de même i seront triés par valeurs de j décroissantes avec la fonction suivante :

```
def tri_insertion_y(liste_points,debut,fin):
    N = len(liste_points)
    for i in range(debut+1,fin+1):
        cle = liste_points[i]
        j = i-1
        while j>=debut and liste_points[j][1] < cle[1]:
            liste_points[j+1] = liste_points[j]
            j = j-1
        liste_points[j+1] = cle
```

Finalement, la fonction suivante repère dans une liste triée par valeur de i croissantes les points de même i et les trie :

```
def tri_y(liste_points):
    N=len(liste_points)
    debut=0
    fin=debut
    for i in range(1,N):
        if liste_points[i][0]==liste_points[i-1][0]:
            fin +=1
        else:
            if fin > debut:
                tri_insertion_y(liste_points,debut,fin)
            debut = fin+1
            fin = debut
```

Voici les deux fonctions permettant d'obtenir l'enveloppe convexe d'une liste de points :

```
def condition(P1,P2,M):
    return (M[0]-P1[0])*(P2[1]-P1[1])-(M[1]-P1[1])*(P2[0]-P1[0])>0

def enveloppe_convexe(liste_points):
    N=len(liste_points)
    env = [liste_points[0],liste_points[1]]
    for i in range(2,N):
        env.append(liste_points[i])
        valide = False
        while not(valide) and len(env)>2:
```

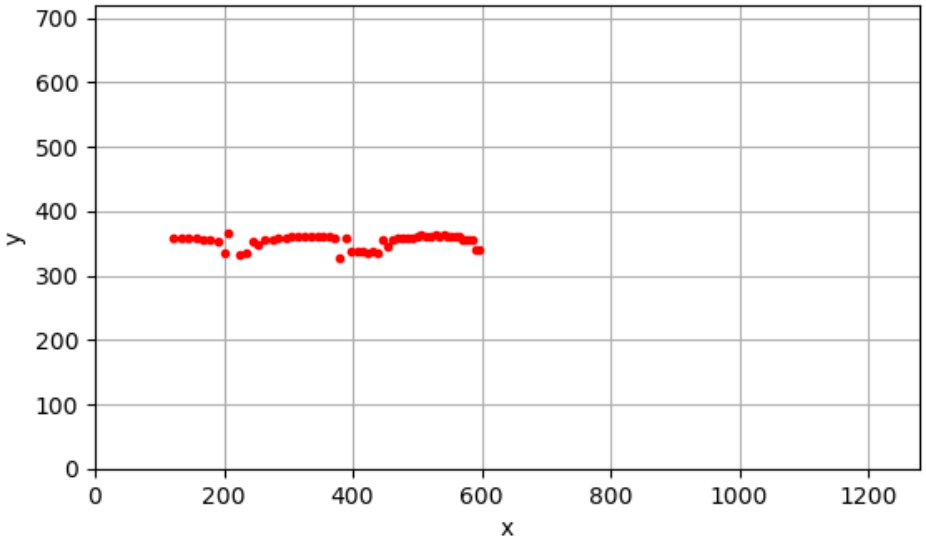
```
            P3 = env.pop()
            P2 = env.pop()
            P1 = env.pop()
            if condition(P1,P2,P3):
                env.append(P1)
                env.append(P2)
                env.append(P3)
                valide = True
            else:
                env.append(P1)
                env.append(P3)
    env.append(liste_points[N-2])
    for i in range(N-3,-1,-1):
        env.append(liste_points[i])
        valide = False
        while not(valide) and len(env)>2:
            P3 = env.pop()
            P2 = env.pop()
            P1 = env.pop()
            if condition(P1,P2,P3):
                env.append(P1)
                env.append(P2)
                env.append(P3)
                valide = True
            else:
                env.append(P1)
                env.append(P3)
    return env
```

Voici la boucle de traitement, qui génère une vidéo où sont tracées les pixels du bord de l'objet (en vert) et l'enveloppe convexe (en bleu).

```
i1=382
j1=118

(M,N,c)=frame.shape
liste_G1 = []
f=0
trace=True

videoWriter = cv.VideoWriter("cylindre-1-positions.avi",cv.VideoWriter_fourcc('I','4','2

while success:
    success,frame=video.read()

    hsv = cv.cvtColor(frame,cv.COLOR_BGR2HSV)

    back_proj = cv.calcBackProject([hsv],[0,1],hist1,[0,180,0,255],scale=1)
    r,img = cv.threshold(back_proj,140,255,cv.THRESH_BINARY)

    img = cv.dilate(img,element)
    img = cv.erode(img,element)
    [i1,j1] = recherche_voisinage(img,M,N,i1,j1)
    liste_pixels = extraction_pixels_bord_objet(img,M,N,i1,j1)
    tri_insertion_x(liste_pixels)
    tri_y(liste_pixels)
```

```
    enveloppe = enveloppe_convexe(liste_pixels)
    liste_pixels=numpy.array(liste_pixels)
    A = ((liste_pixels[:,1]).min(),(liste_pixels[:,0]).min())
    B = ((liste_pixels[:,1]).max(),(liste_pixels[:,0]).max())
    [xG,yG]=[(A[0]+B[0])/2,(A[1]+B[1])/2]
    j1=int(xG)
    i1=int(yG)
    liste_G1.append([xG,M-yG])
    L=50
    img[i1-L:i1+L,j1]=200
    img[i1,j1-L:j1+L]=200
    if trace:
        cv.imshow('frame',img)
        if cv.waitKey(500)==113: break
    for [i,j] in liste_pixels:
        frame[i,j]=[0,255,0]
    if enveloppe:
        for k in range(len(enveloppe)-1):
            p1 = enveloppe[k]
            p2 = enveloppe[k+1]
            cv.line(frame,(p1[1],p1[0]),(p2[1],p2[0]),[255,0,0],2)

    frame[i1-L:i1+L,j1]=[255,255,255]
    frame[i1,j1-L:j1+L]=[255,255,255]
    cv.imshow('frame',frame)
    videoWriter.write(frame)
    if cv.waitKey(500)==113: break
    f+=1

videoWriter.release()
```

Voici les positions obtenues à partir de l'enveloppe rectangulaire :



La position est légèrement décalée vers le bas lorsque la partie orange détectée ne s'étend pas jusqu'en haut du cylindre, ce qui d'ailleurs permet d'accéder à la vitesse de rotation du cylindre.