VISION

-

COULEUR & TRACKING

Florent Muret

Jean-Baptiste Rambaud

Introduction

Ce projet s'intéresse au tracking d'une boule de couleur par un drone ce qui consiste à identifier un objet présent sur une vidéo et de pouvoir suivre ce même objet si ce dernier bouge à l'image. Ce tracking est possible grâce à un programmes d'analyse d'images qui utilisent la couleur d'un objet pour le localiser en temps réel à l'aide d'une caméra.

Dans ce projet, nous étudierons dans un premier temps un algorithme ayant les caractéristiques décrit précédemment, puis nous améliorons par celui-ci afin d'inclure une détection de forme de l'objet. Enfin nous mettrons en œuvre un programme permettant de détecter un objet présent au centre de l'image.

Question 1

Nous commençons par tester le code **python2 balls_tracker.py** avec python v2 sur ball.mp4 puis ball2.mp4.

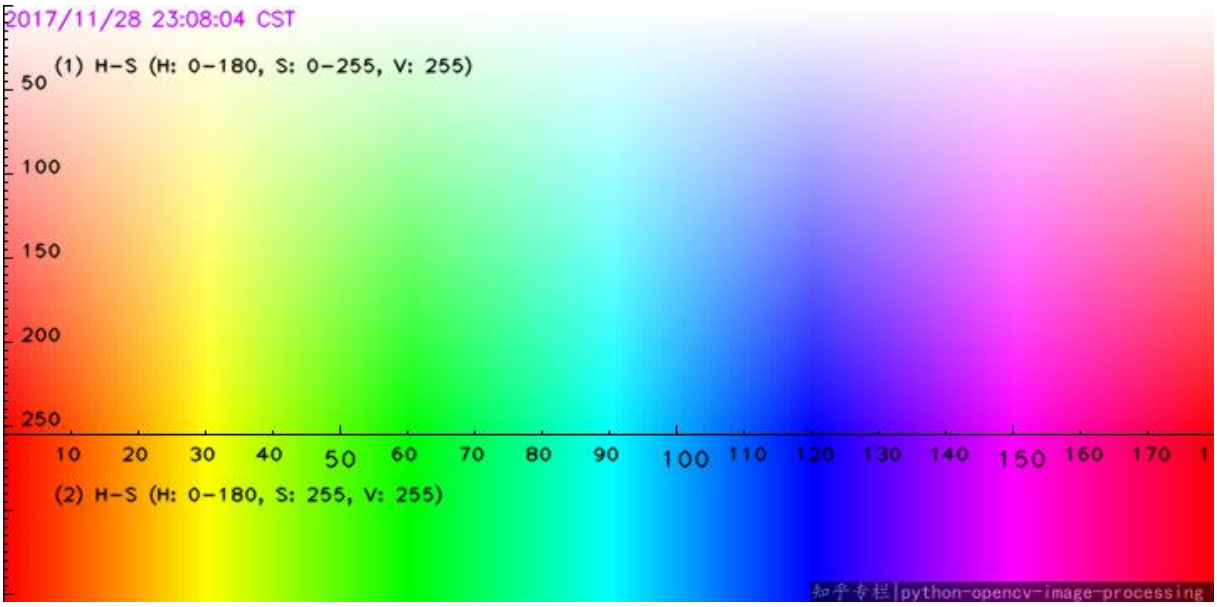
On peut changer la vidéo que l'on choisit en modifiant l'argument que l'on passe à la commande "cv2.VideoCapture()"

Suites à nos observations, on peut en déduire que l'objectif de ce code est de traquer des objets en fonction de la couleur définie dans le masque.

Le tracker prend la forme d'un point qui se place au centre de la zone qui correspond à la couleur que l'on souhaite détecter. Si l'on met deux objets de même couleur, il y aura conflit.

Question 2

Le modèle HSV (*Hue Saturation Value* ou HSB pour *Hue Saturation Brightness*), est un système de gestion des couleurs en informatique. Il est basé sur les trois composantes définies par une approche psychologique et perceptuelle de la couleur : teinte, saturation et valeur.



Cette photo nous permet de paramétrer les paramètres de threshold de notre tracking.

Les coefficients de threshold choisis sont :

- Blue :
  - Lower = [95,70,70]
  - Upper = [100,200,200]
- Pink :
  - Lower = [170,120,120]
  - Upper = [175,200,200]
- Yellow :
  - Lower = [27,70,70]

- Upper = [35,200,200]

- Green :
  - Lower = [40,70,70]
  - Upper = [80,200,200]

Nous avons décidé d'augmenter l'intensité et la valeur à 120 pour la couleur Pink pour ne pas détecter la main par erreur.

*Tracking balle **Jaune***

*Tracking balle **Bleue***

*Tracking balle **Rose***

*Tracking balle **Verte***

Le code permettant d'effectuer les trackings précédent est fourni ci-dessous :

```
import cv2
import numpy as np

# For OpenCV2 image display
WINDOW_NAME = 'GreenBallTracker'

color = 'yellow'


def track(image):

    '''Accepts BGR image as Numpy array
       Returns: (x,y) coordinates of centroid if found
                (-1,-1) if no centroid was found
                None if user hit ESC
    '''

    # Blur the image to reduce noise
    blur = cv2.GaussianBlur(image, (5,5),0)

    # Convert BGR to HSV
    hsv = cv2.cvtColor(blur, cv2.COLOR_BGR2HSV)

    # Threshold the HSV image for only green colors
    if color == 'green' :
      lower_array = np.array([40,70,70])
      upper_array = np.array([80,200,200])

      # Threshold the HSV image for only blue colors
    if color == 'blue' :
      lower_array = np.array([95,70,70])
      upper_array = np.array([100,200,200])

    if color == 'yellow' :
      lower_array = np.array([27,70,70])
      upper_array = np.array([35,200,200])

    if color == 'pink' :
      lower_array = np.array([170,120,120])
      upper_array = np.array([175,200,200])



    # Threshold the HSV image to get only green colors
    mask = cv2.inRange(hsv, lower_array, upper_array)

    # Blur the mask
```

Code pour le tracking d'une balle de couleur Jaune

On aurait pu pour plus de propretés, passer l'argument de couleur en ligne de commande dans le terminal et le récupérer à l'aide de argv.

Question 4

Le code autopylot_agent.py permet au drone de garder l'objet en visuel. Ainsi si l'objet se déplace le drone le suivra.

Pour cela le programme calcul à chaque itération le centre de la balle.

```
# Grab centroid of green ball
ctr = greenball_tracker.track(image)
```

Il calcul ensuite l'erreur et grâce à un contrôle PID il se replace.

```
# Use centroid if it exists
if ctr:

    # Compute proportional distance (error) of centroid from image center
    errx =  _dst(ctr, 0, img_width)
    erry = -_dst(ctr, 1, img_height)

    # Compute vertical, horizontal velocity commands based on PID control after first iteration
    if action.count > 0:
        phi = _pid(action.phi_1, errx, action.errx_1, Kpx, Kix, Kdx)
        gaz = _pid(action.gaz_1, erry, action.erry_1, Kpy, Kiy, Kdy)

    # Remember PID variables for next iteration
    action.errx_1 = errx
    action.erry_1 = erry
    action.phi_1 = phi
    action.gaz_1 = gaz
    action.count += 1
```

Pour se replacer, il récupère la taille de l'image, la divise par deux. Il retourne alors la distance en pourcentage à laquelle il se trouve du centre.

```
# Simple PID controller from http://www.control.com/thread/1026159301
def _pid(out_1, err, err_1, Kp, Ki, Kd):
    return Kp*err + Ki*(err+err_1) + Kd*(err-err_1)

# Returns proportional distance to image center along specified dimension.
# Above center = -; Below = +
# Right of center = +; Left = 1
def _dst(ctr, dim, siz):
    siz = siz/2
    return (ctr[dim] - siz) / float(siz)
```

Le lien avec greenball_tracker.py est que ces deux scripts permettent de traquer une un objet. Leur mode de fonctionnement est cependant différent.

Question 5

Si 2 boules de mêmes couleurs sont placées devant le drone, ce dernier va hésiter, il ne saurait pas quelle boule traquer et donc on verra une alternance du tracker, c'est-à-dire que le point oscillera entre deux positions.

Ce comportement n'est pas pertinent car le tracker devrait rester fixer sur un seul objet.

Le code fourni permet de résoudre ce problème en changeant le comportement du traqueur.

Le nouveau comportement du drone est de faire une détection d'objet et de classifier tous les objets qui répondent au critère de couleur dans un même tableau.

Si 2 boules de mêmes couleurs sont placées devant lui alors le drone choisira de traquer l'objet prenant le plus de place à l'image.

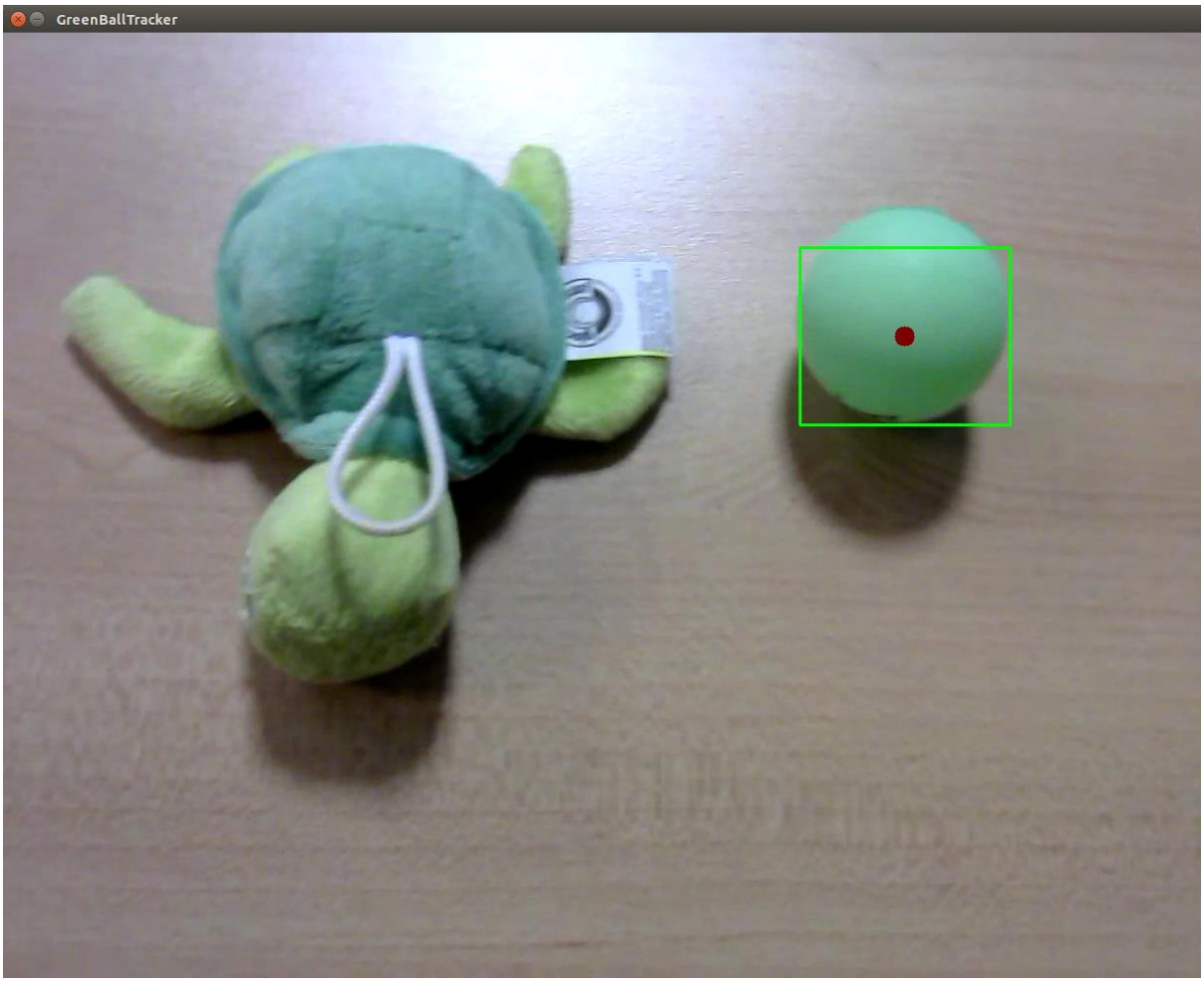
Question 6

Pour obtenir un comportement équivalent, nous avons réutilisé les morceaux de codes précédents afin de faire une détection en fonction de la couleur ainsi que de la taille. Ainsi on va sélectionner le contour le plus grand correspondant aux critères de couleurs.

On ne détecte plus seulement la couleur de la balle mais également sa forme ce qui permet de ne pas avoir de problème si un objet de la même couleur (cf tortue) rentre dans le champ de la caméra.

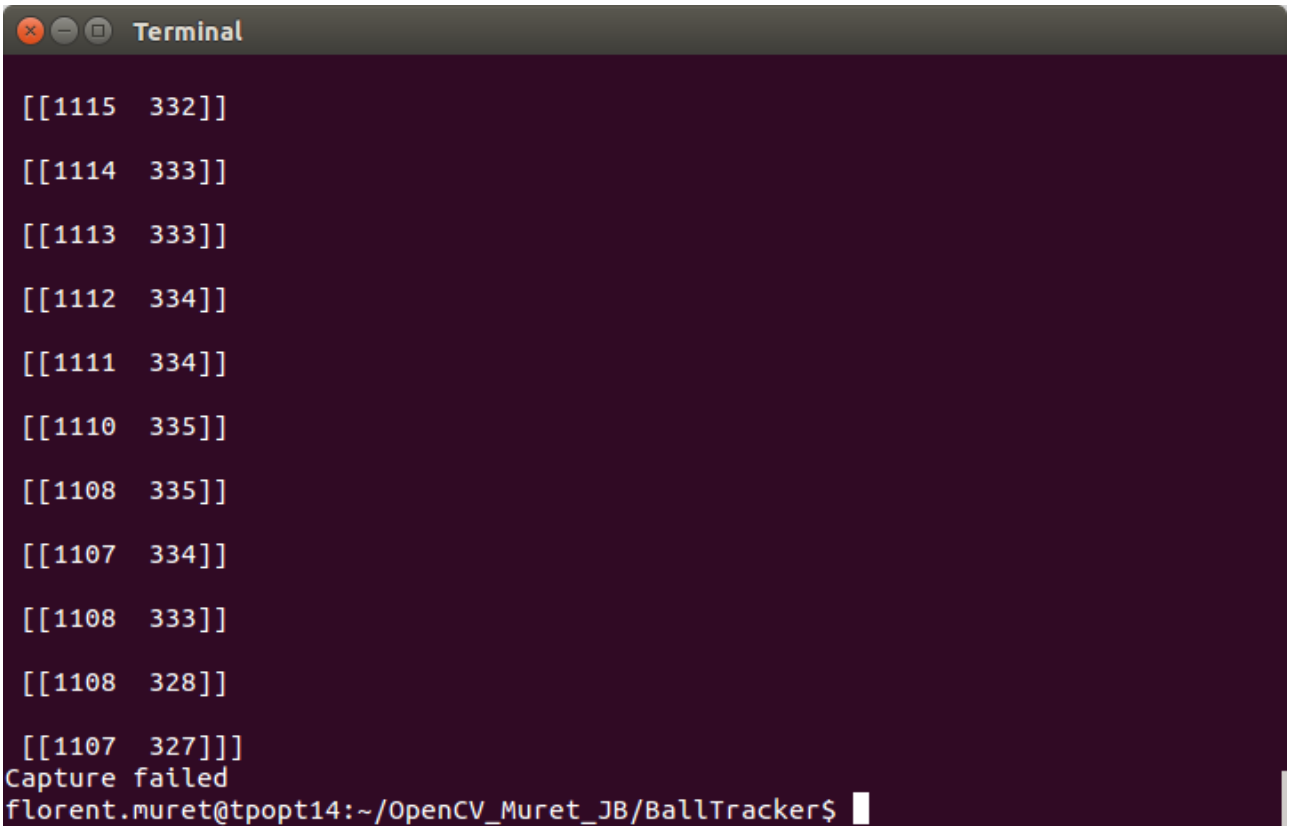
```
#==================== mon code contour ======================

    #on a capture.read en argument = image, ici on utilise bmask
    contours, hierarchy = cv2.findContours(bmask,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)

    showingCNTs = [] # Contours that are visible
    areas = [] # The areas of the contours

    for cnt in contours:
      approx = cv2.approxPolyDP(cnt,0.1*cv2.arcLength(cnt,True),True)

      area = cv2.contourArea(cnt)
      if area > 200:
  areas.append(area)
  showingCNTs.append(cnt)

        ## Only Highlight the largest object
    if len(areas)>0:
      m = max(areas)
      maxIndex = 0
      for i in range(0, len(areas)):
  if areas[i] == m:
    maxIndex = i
    cnt = showingCNTs[maxIndex]

      cv2.drawContours(image,[cnt],0,(0,0,255),-1)

      x,y,w,h = cv2.boundingRect(cnt)
      cv2.rectangle(image,(x,y),(x+w,y+h),(0,255,0),2)

      center = (x+(w/2),y+(h/2))
      print(center)
      cv2.circle(image,center, 10, (0,0,125), -1)

    else :
      center = (-1,-1)

# ============ d'autre contour et moyen d'entourer =================
```

Dans le code ci-dessus, on effectue la sélection par taille des contours. En effet, on récupère tous les contours détectés dans un tableau, et on prend la valeur maximale en termes de superficie de ce tableau à afficher. On dessine ensuite le point rouge ainsi qu'un rectangle vert entourant la forme.



On peut voir le résultat ci-dessus. La tortue n'est ici pas considérée comme la forme la plus grosse à cause des critères de vert que nous avons rentrée. En effet, nous avons recherché du vert plus fluo, c'est pour cela que le détecteur ne prend pas en compte la carapace. Nous nous sommes par la suite amusés à tester les non pas la valeur maximale du tableau, mais les deuxième, troisièmes plus grandes valeurs, et il s'avère que le détecteur entoure les différentes pattes de la tortue.

En réalité, une grosse partie de la détection va dépendre de la largeur de bande que nous allons donnée au tableau des couleurs (lower_array & upper_array), comme nous avons pu le voir dans les questions précédentes.



On peut voir ci-dessus que le script python renvoie les coordonnées du centre de l'objet le plus grand qu'il trouve dans sa détection, soit ici la balle.

Question 7

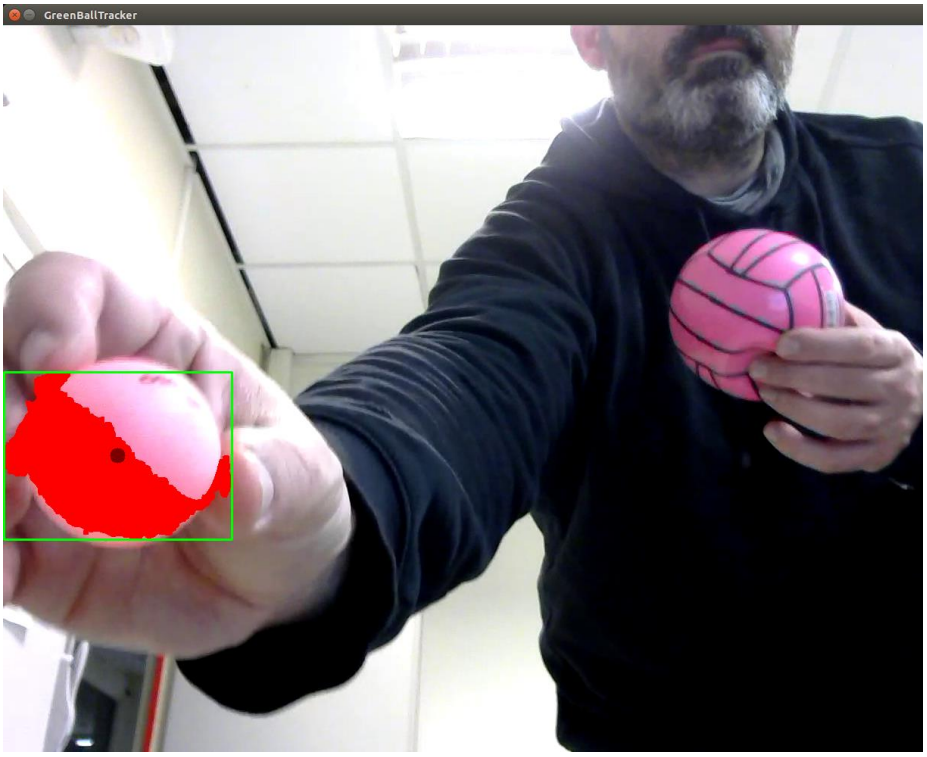
Nous cherchons maintenant à faire fonctionner notre code en prenant en compte la couleur rose présente sur la vidéo, afin de traquer la balle rose la plus grosse. Sur cette vidéo, il y a deux balles rose, et donc selon la position de celles-ci, l'une sera plus grosse que l'autre et inversement au fur et à mesure de la vidéo.

On utilise ici le même code que précédemment, tout en changeant la valeur du filtre de couleur, nous avons pris ici :

lower_array = np.array([160,120,120])
upper_array = np.array([200,255,255])

Cette valeur est importante car selon le seuil que nous fixons, nous nous sommes rendus compte que nous pouvions détecter seulement une seule balle, ou bien l'autre.

Détection de la première balle



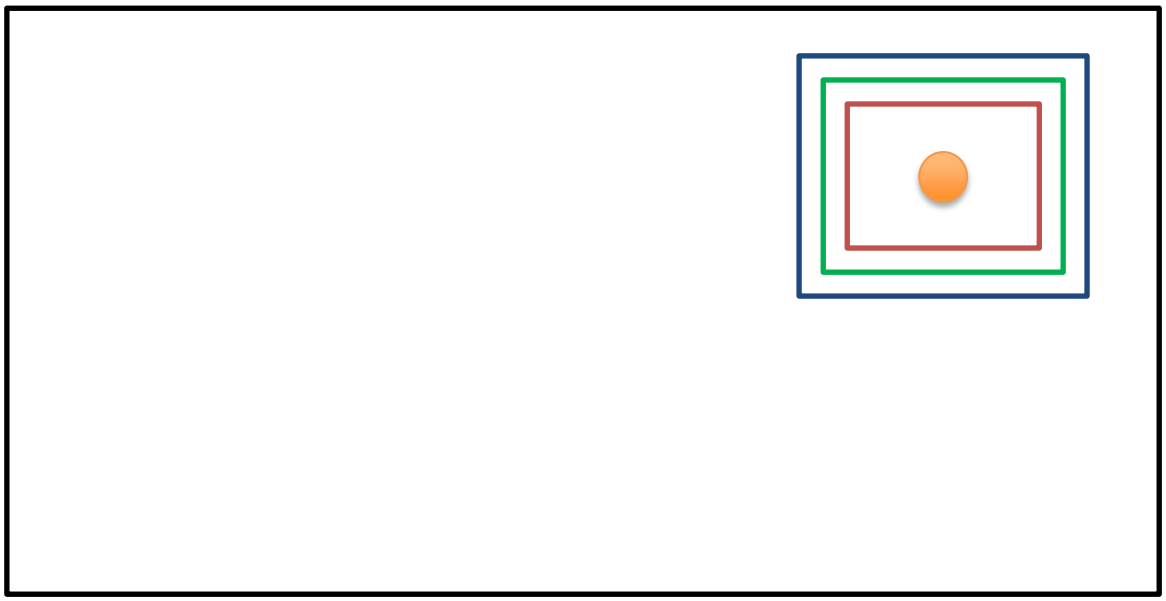
Détection de la seconde balle

Sur ces deux captures d'écrans on peut voir que nous détectons à la suite la grosse puis la balle de ping pong en fonction de ou l'une se situe sur l'écran. On a également rajouté une multitude de points rouges sur la zone détecté.

Question 8

On peut procéder à l'apprentissage d'une couleur précise. C'est-à-dire que même si deux balles sont identiques, pour l'ordinateur, elles auront quand même des nuances en HSV. On peut donc sauvegarder cette valeur de couleur, avec un certain seuil, et remodifier cette valeur à chaque boucle (l'éclairage peux changer la valeur de HSV).

On peut sinon raisonner de proche en proche. En effet, en partant du principe que le premier objet détecté est le bon (sachant qu'il doit respecter quand même certains critères, c'est-à-dire la taille, la couleur), on peut en déduire que celui-ci va se trouver dans une zone proche de la ou il a été détecté précédemment. On peut donc réduire la zone de recherche autour du point précédent à une certain carré, afin de savoir ou l'objet précédent à bouger. Si l'objet à bougé trop vite ou a été masqué, on élargit progressivement la zone de recherche. Sur ce schéma ci-dessous, on peut voir en orange l'objet qui à été détecté en premier, ou alors en position (n-1). Le carré rouge serait la première zone de recherche, le carré vert la seconde, au cas ou le dis objet correspondant aux mêmes critères de couleur et taille ne soit toujours pas trouvé, en bleu la zone de recherche suivante, et ainsi de suite. On peut définir l'étendue de la zone de recherche en fonction du nombre de FPS de la vidéo, et du temps de traitement de notre algorithme.



Le plus efficace serait évidement une combinaison de ces deux méthodes.

Question 9

On met ici en place la technique de proche en proche abordé dans la question précédente.

```
color = 'pink'
center = (0,0)


def track(image):

  dim = image.shape
  h=dim[0]
  w=dim[1]

    if center != (0,0):
      for i in range(1, 25):
        first, second = center  #position du centre

        plage_x = i*25    #largeur de la fenetre de recherche
        plage_y = i*25    #longueur de la fenetre de recherche

        first  = first - (plage_x/2)  #position extreme en x
        second = second - (plage_y/2) #position extreme en y

        longuer_max_x = first + plage_x
        longuer_min_x = first
        longuer_max_y = second + plage_y
        longuer_min_y = second

        if first+plage_x > w : #si la fenetre dépasse en x en positif
          longuer_max_x  = w
        if first+plage_x < 0 : #si la fenetre dépasse en x en negatif
          longuer_min_x  = 0
        if first+plage_y > h : #si la fenetre dépasse en y en positif
          longuer_max_y  = h
        if first+plage_y < 0 : #si la fenetre dépasse en y en negatif
          longuer_min_y  = 0

        image = image[ second:second+plage_y, first:first+plage_x]

```

Dans le code ci-dessus, on ajuste la fenêtre de recherche en rajoutant le code suivant. Avant de mettre en place cette méthode de proche en proche, on attend que l'algorithme ait détecté un premier contour (c'est-à-dire récupéré un centre différent de (0,0)), assez gros pour qu'il ait du sens. Puis, une fois que l'on a récupéré la coordonnée du centre du premier objet détecté (que nous allons traquer), on effectue une recherche par zone, au début de 25 pixels, puis de 25*2, 25*3, etc, jusqu'à ce que la zone couvre toute l'image (au cas où l'objet ait bougé rapidement). Ainsi on s'assure d'avoir suivi le même objet.

Question 10

Pour mettre en place un apprentissage de la couleur d'un objet se trouvant au milieu de l'image nous commençons par récupérer les coordonnés du pixel du milieu de l'image grâce aux dimensions de la fenêtre.

```
# Test with input from camera
if __name__ == '__main__':

    capture = cv2.VideoCapture('ball2.mp4')

    while True:

        okay, image = capture.read()
  height = np.size(image, 0)
  width = np.size(image, 1)
  center_img = ((width /2 ) , (height/2 ) )

        if okay:
            if not track(image):
                break
            if cv2.waitKey(1) & 0xFF == 27:
                break
        else:
            print('Capture failed')
            break
```

Une fois ces coordonnées obtenues, nous déterminons les niveaux hsv du pixel du milieu et des pixels qui lui sont adjacents.

On procède ensuite à un moyennage autour de ce pixel, afin de ne pas se limiter à un seul pixel. Plus le moyennage sera grand, plus la détection sera précise.

Nous pouvons alors utiliser ce niveau hsv moyenné pour notre détection de l'objet situé au centre de l'image (upper and lower array).

Cependant cette méthode a des limites car si l'objet ne se situe pas exactement au milieu de l'image, l'algorithme ne sera pas en mesure de le détecter. C'est donc à nous de s'adapter et de trouver la bonne position.

```
    # Blur the image to reduce noise
    blur = cv2.GaussianBlur(image, (5,5),0)

    # Convert BGR to HSV
    hsv = cv2.cvtColor(blur, cv2.COLOR_BGR2HSV)
    global index
    sum_h = 0
    sum_s = 0
    sum_v = 0

    if index == 1:
      center_array = [center_img,
          (center_img[0],center_img[1]+1),
          (center_img[0]+1,center_img[1]+1),
          (center_img[0]+1,center_img[1]),
          (center_img[0]-1,center_img[1]-1),
                (center_img[0]-1,center_img[1]),
                (center_img[0],center_img[1]-1),
                (center_img[0]-1,center_img[1]+1),
                (center_img[0]+1,center_img[1]-1)]

      for i in range(0,len(center_array)):
px = hsv[center_array[i]]
pixel_table.append(px)


sum_h = sum_h + pixel_table[i][0]
sum_s = sum_s + pixel_table[i][1]
sum_v = sum_v + pixel_table[i][2]

      sum_h = sum_h/len(center_array)
      sum_s = sum_s/len(center_array)
      sum_v = sum_v/len(center_array)

      print(sum_h,sum_s,sum_v)
      #print(hsv[center_img])
      if color == 'detection' :
lower_array = np.array([sum_h,sum_s,sum_v])
upper_array = np.array([sum_h+20,200,200])

      index+=1
```

Ci-dessus le reste du code permettant d'effectuer le moyennage de la couleur de l'objet se trouvant au centre.

Question 11

En connaissant la taille de la balle, on connait selon l'algorithme de traitement que nous avons fait précédemment la position de son centre en fonction de la taille de l'image. On a donc une position de centre en pixel, ainsi que la position des contours grâce à notre carre qui entoure cette balle, de ce fait ses dimensions. Ainsi, on peut faire varier notre valeur de repère en fonction de ces valeurs de pixels : en effet, si la balle fait un rayon de 15cm par exemple, et que notre carré qui l'entoure fait 85 pixels, alors on a 15cm = 85 px, soit 1 pixel = 0.17 cm, et on peut donc placer notre balle dans un repère cartésien. Les dimensions vont bien sur évoluer en fonction de la perspective, ce qui va faire évoluer la position de la balle.

Nous n'avons pas eu le temps de coder cette partie.

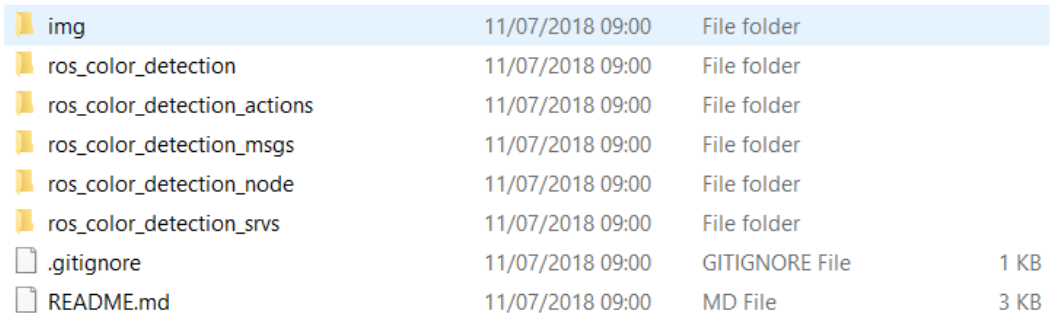
Question 12

Si l'on sait à l'avance que l'objet est une balle, alors grâce aux données de la détection, on sait par avance que l'objet est rond, ce qui veux dire que l'on peut compléter la détection de contour, avec un remplissage. En effet, parfois la détection

à ses limites et cela est dû par exemple à des facteurs comme l'éclairage. On peut le voir ci-contre, en effet la balle n'est pas totalement recouverte de point rouge, c'est-à-dire qu'une partie a été détecté mais il y a une partie manquante. Il est possible de « compléter » cette partie manquante en effectuant un balayage circulaire au centre de l'objet détecté, avec un rayon ayant pour valeur le point extrême, c'est-à-dire celui le plus éloigné du centre. Ainsi on peut reconstituer le cercle autour de la balle, même si la détection ne se fait pas complètement.

Question 13

On commence par récupérer le code, afin de l'analyser.



Le dossier img contient les images traitées, les autres dossiers contiennent le Cmake list, la définition du service ainsi que des messages. Afin de savoir comment sont représentés les couleurs on peut regarder le format des trames de msg :

```
 string color_name
# color name without correction (black, white, gray)
string color_temp
# closest web color name
string color_web
# custom black, white, gray, dark brightness name
string color_brightness_name
# rgb[0]=R, rgb[1]=G, rgb[2]=B
float32[] rgb
# hsv[0]=H, hsv[1]=S, hsv[2]=V
float32[] hsv
# percentage of the current color into the image (after kmean clusterisation)
float32 percentage_of_img
```

Le code ci-dessus provient du message ColorD.msg, le second message ColorDlist.msg étant seulement une liste d'argument de type ColorD. On peut voir ici que le message va contenir comme information plusieurs chaine de caractère afin de désigner la couleur et la luminosité. On va également transmettre les valeurs de RGB ainsi que de HSV sous forme de tableau, comme marqué en commentaire, ainsi que le pourcentage de l'image qui contient cette couleur. Ces valeurs seront complété dans cette partie de code dans le script python :

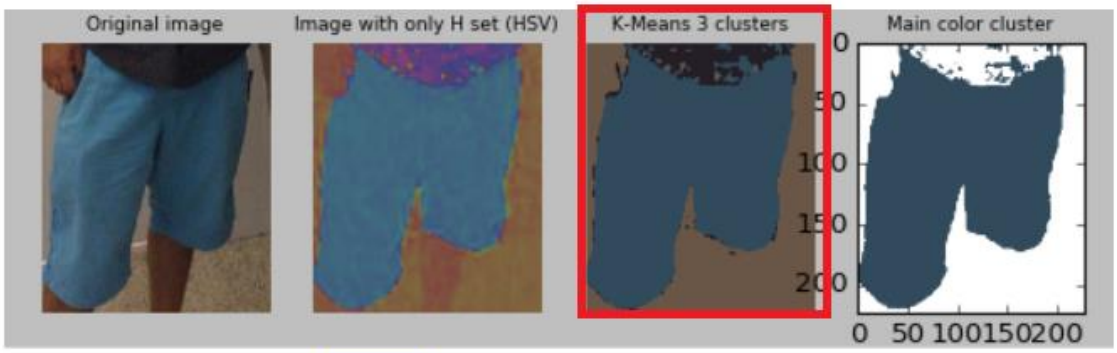
```
colorD=ColorD()
colorD.color_name=str(color_name_result)
colorD.color_web=str(closest_name_original_value)
colorD.color_temp=str(simpleColorFromOriginal)
colorD.color_brightness_name=str(label_brightnes)
colorD.rgb=[R,G,B]
colorD.hsv=[cluster.getValue()[0],cluster.getValue()[1],cluster.getValue()[2]]
colorD.percentage_of_img=float(cluster.getNbPixel())/float(pixelSum)
return colorD
```

Nous allons nous intéresser au code du nœud, dans le dossier Node. Ouvrons le fichier ColorDetectionNode.py.

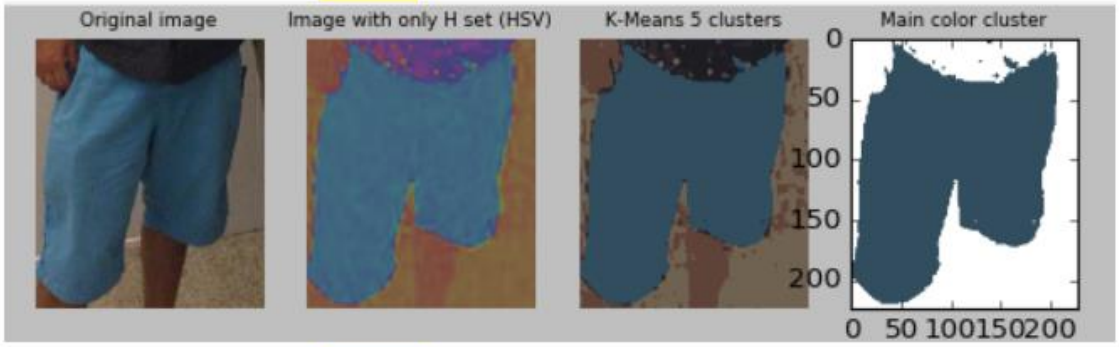
Le code fonctionne via un service 'detect_color_srv ', et le principe de fonctionnement est le suivant : on convertit tout d'abord l'image en hsv, puis on utilise seulement la teinte de l'image dans le modèle HSV (soit seulement H), afin d'homogénéiser l'image. Par la suite, on effectue le traitement avec le paramètre K_mean (nombre de couleur dominante à extraire), qui consiste à effectuer un moyennage des couleurs retenues, c'est pour cela que par exemple la valeur dominante du short est un bleu sombre. Plus le K_mean sera grand, plus on recherchera un nombre de couleur importantes, et de ce fait il y aura des séparations dans l'homogénéisation. Par exemple, lorsque K_mean = 10, puisque l'algorithme cherche 10 couleurs dominantes à extraire, certaines parties sombres du short vont être associées à une autre couleur que du bleu, et donc lors du rendu final, c'est-à-dire lorsque l'on va éliminer les 9 couleurs non dominantes, alors ces zones du shot dites « sombres » vont être manquante.

En résumé si l'on prend K_mean petit, alors le moyennage retournera plus de zone mais cela sera moins précis.

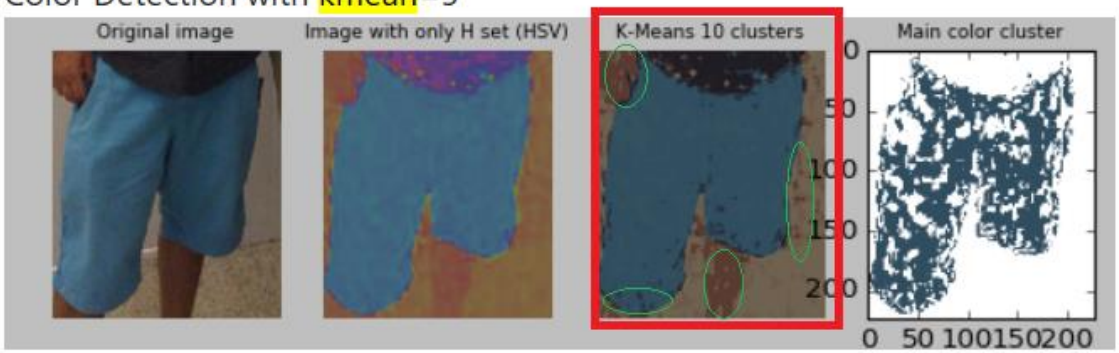
On peut noter ces petites différences en fonction de K_mean sur l'image ci-dessous.



Color Detection with kmean=3



Color Detection with kmean=5



Color Detection with kmean=10

Au niveau du fonctionnement ROS, le nœud va souscrire à un topic qui est l'image que l'on souhaite traiter, ici l'image du short. Le nœud va ensuite publier sur un topic /image_color, qui va renvoyer le message ColorDlist que nous avons vu précédement. Le code met donc en place un service :

```
#declare ros service

self.detectColorSrv = rospy.Service('detect_color_srv', DetectColorFromImg, self.detectColorSrvCallback)
```

On choisit les paramètres de notre détection via common_color.yaml qui contient comme paramètre le K_mean, le path de l'image et l'affichage (ou non car bool).

Conclusion

Dans ce projet nous avons étudié plusieurs méthodes afin de mettre en place une détection d'objet. La première consiste à détecter un objet en fonction de sa couleur. La seconde permet d'identifier un objet en caractérisant sa forme. La dernière permet de détecter un objet situé au centre de l'image.

Cependant d'autres méthodes existent pour la détection d'image. Une méthode que l'on pourrait envisager serait d'entraîner un réseau de neurone à reconnaître un ou des objets en lui fournissant une base de données sur laquelle ce réseau pourrait apprendre grâce au mécanisme de back propagation. Un avantage de cette méthode est que le réseau de neurone ne reconnaîtra pas l'objet uniquement grâce à sa couleur ou à sa forme mais grâce à une combinaison multiple de facteurs. Cependant pour arriver à un résultat correct il faut avoir à sa disposition une base de données importante.