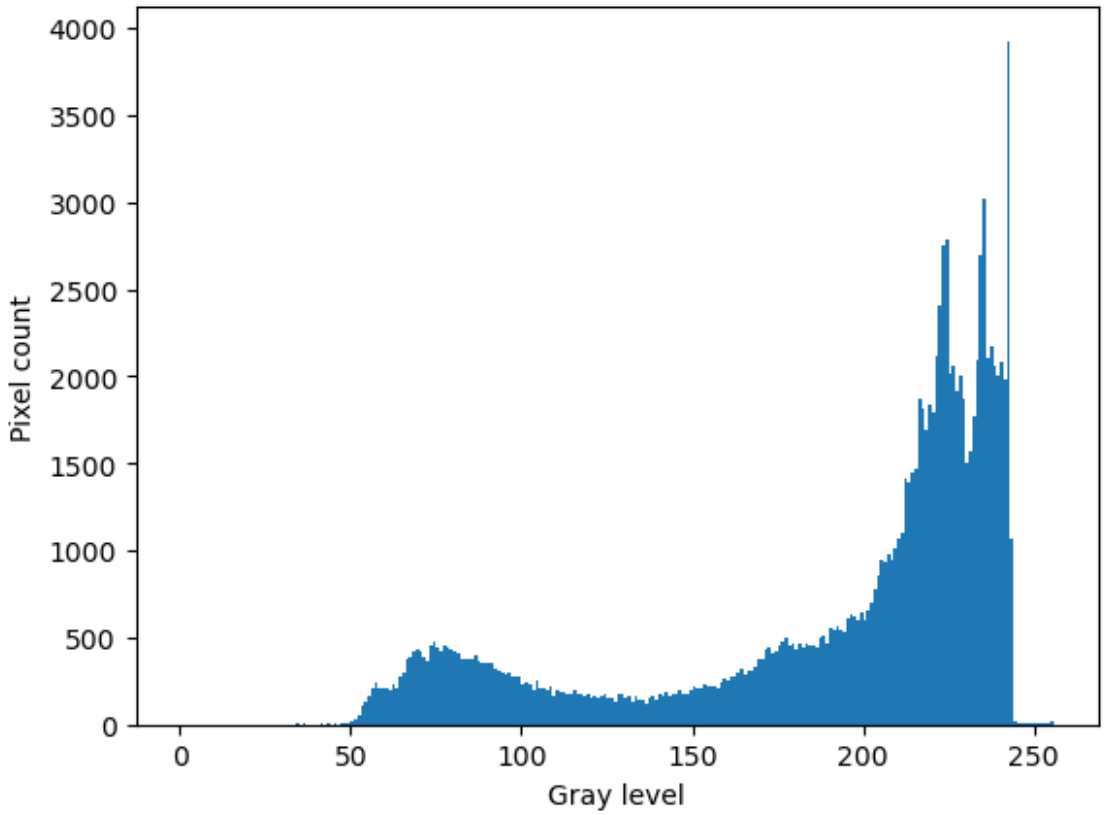
```
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

# Es 2.1

```
img = cv2.imread('redberry_bw.png', cv2.IMREAD_GRAYSCALE)

plt.imshow(img, cmap='gray')
plt.title('Original image in B/W')
plt.axis('off')
plt.show()
```


The selected image represent berries in an human hand, we want to segment the berries.

```
plt.hist(img.ravel(),256,[0,256])
plt.xlabel("Gray level")
plt.ylabel("Pixel count")
plt.show()
```




We can see a valley between 100 and 150, so we can try to put there the threshold. We can apply this threshold with the openCV function.

```
ret1,th_manual = cv2.threshold(img,125,255,cv2.THRESH_BINARY)

plt.imshow(th_manual, cmap='gray')
plt.title('Manual segmentation with threshold: ' + str(ret1))
plt.axis('off')
plt.show()
```


As we can see, it colours the berries in blacks, we have some imprecision due to reflections in the berries and shadows. (a closing operation would be useful I think)

Now we try with Otsu, so we don't need to choose a threshold but the algorithm will calculate it measuring the intra-class variance.

```
ret2,th_otsu =
cv2.threshold(img,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)

plt.imshow(th_otsu, cmap='gray')
plt.title('Otsu segmentation with threshold: ' + str(ret2))
plt.axis('off')
plt.show()
```


Looking at the image I would say that it takes more unwanted detail in respect to the first method.

Here I define two functions: one for calculating the dice coefficient and one for taking the red region in the ground truth image (the corrected segmentation of the image)

```
def dice_coefficient(img1,img2):
    intersection = np.logical_and(img1, img2)
    dice = (2*np.sum(intersection))/(np.sum(img1) + np.sum(img2))
    return dice

def extract_red_mask(image):
    lower_red = np.array([0, 50, 50])
    upper_red = np.array([10, 255, 255])

    hsv_image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

    red_mask = cv2.inRange(hsv_image, lower_red, upper_red)
    inverted_mask = cv2.bitwise_not(red_mask)

    return inverted_mask

ground_img = cv2.imread('redberry_seg.png')

red_mask = extract_red_mask(ground_img)
```

```
plt.imshow(red_mask, cmap='gray')
plt.title('Ground truth segmentation')
plt.axis('off')
plt.show()
```


For calculating the dice coefficient I want binary values, so instead of 0 or 255 -> 0 or 1

```
manual_bin = np.where(th_manual > 0,1,0)
otsu_bin = np.where(th_otsu > 0,1,0)
ground_bin = np.where(red_mask > 0,1,0)

dice_otsu = dice_coefficient(otsu_bin,ground_bin)
dice_otsu
```

```
0.9681814361947981
```

```
dice_manual = dice_coefficient(manual_bin,ground_bin)
dice_manual
```

```
0.9745732447682308
```

The dice coefficients are both high, but as it's possible to see from the images above, the manual setting of the threshold resulted in a better result.

Sources:

- https://stackoverflow.com/questions/51225657/detect-whether-a-pixel-is-red-or-not/51228567#51228567
- https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html

# Es 2.2

For adapting the tracking object code to the video i registered, I just substitute the coordinates of the object (x,y,width,height) in the first frame of the video, such that the function can extract the first ROI (region of interest) and its histogram.

The calcBackProject generate a probability map, for a given image these are the probabilities that the pixels fits a given histogram of a ROI. So it takes the histogram of the ROI that is selected in the first frame and the probability map is created based on the color informations. This map will be useful for the meanShift.

In this case, it's used the HSV space, because it separates the color information (H) and it's useful for tracking object based on the color.

The meanShift method takes the result of the Back Projection on a ROI, and it moves the track window iteratively towards the region with maximum probability.

So as we saw in the lectures:

- Starts from an initial window
- Computes the centroid
- Move the window toward it
- Repeat until term criteria (max iterations or convergence)

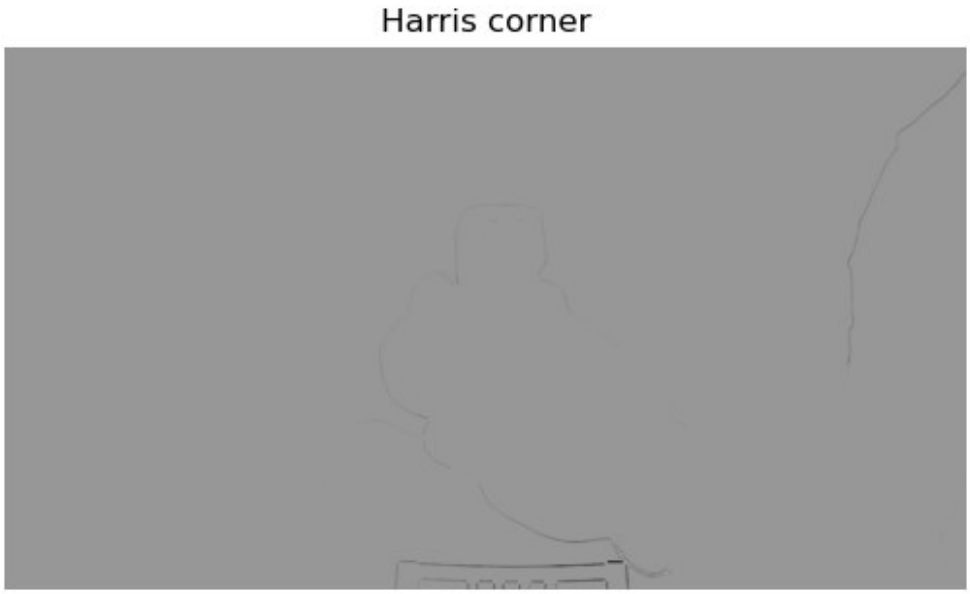
```
video = cv2.VideoCapture("duck2.mkv")
_, first_frame = video.read()
x = 914
y = 410
width = 152
height = 92
roi = first_frame[y: y + height, x: x + width]
hsv_roi = cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)
roi_hist = cv2.calcHist([hsv_roi], [0], None, [180], [0, 180])
roi_hist = cv2.normalize(roi_hist, roi_hist, 0, 255, cv2.NORM_MINMAX)
term_criteria = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10,
1)

while (True):
    success, frame = video.read()
    if success==True:
        hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
        mask = cv2.calcBackProject([hsv], [0], roi_hist, [0, 180], 1)
        _, track_window = cv2.meanShift(mask, (x, y, width, height),
term_criteria)
        x, y, w, h = track_window
```

```
        cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)
        cv2.imshow("Mask", mask)
        cv2.imshow("Frame", frame)
        k = cv2.waitKey(1) & 0xff
        if k == ord('q'):
            break
    else:
        break
video.release()
cv2.destroyAllWindows()
```

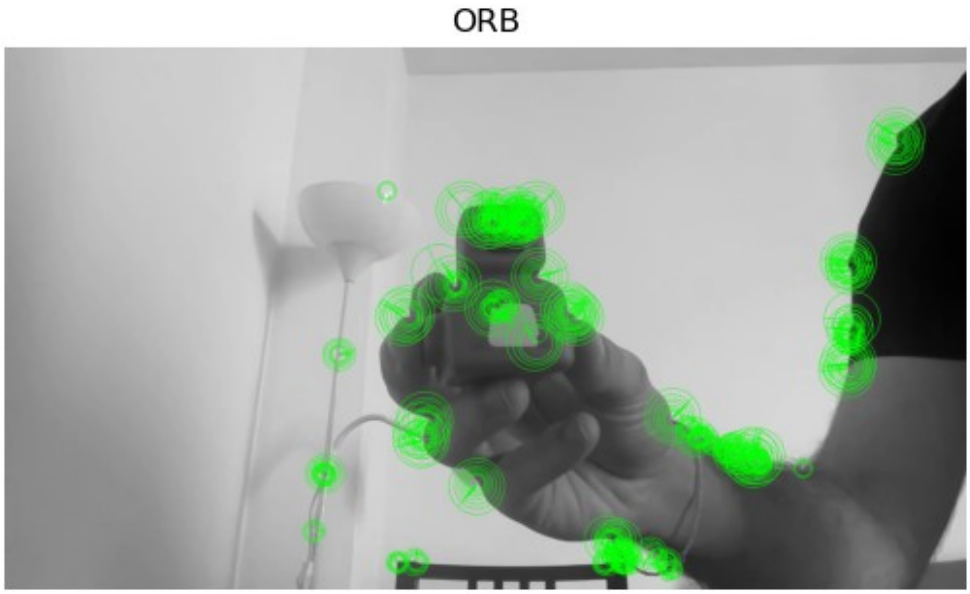
# Es 2.3

I select 2 frames from the video, with different rotation of the object.

```
video = cv2.VideoCapture("duck2.mkv")

#straight
video.set(cv2.CAP_PROP_POS_FRAMES, 1)
_, frame_1 = video.read()
gray1 = cv2.cvtColor(frame_1, cv2.COLOR_BGR2GRAY)
gray1_32 = np.float32(gray1)


#overturned
video.set(cv2.CAP_PROP_POS_FRAMES, 150)
_, frame_2 = video.read()
gray2 = cv2.cvtColor(frame_2, cv2.COLOR_BGR2GRAY)
gray2_32 = np.float32(gray2)

harris = cv2.cornerHarris(gray1_32, 2, 3, 0.04)

sift = cv2.SIFT_create()
orb = cv2.ORB_create()

keypoints_sift= sift.detect(gray1, None)
keypoints_orb = orb.detect(gray1, None)

image_sift = cv2.drawKeypoints(gray1, keypoints_sift, None,
color=(0,255,0), flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
image_orb = cv2.drawKeypoints(gray1, keypoints_orb, None,
color=(0,255,0), flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

plt.imshow(harris, cmap='gray')
plt.title('Harris corner')
plt.axis('off')
plt.show()
```



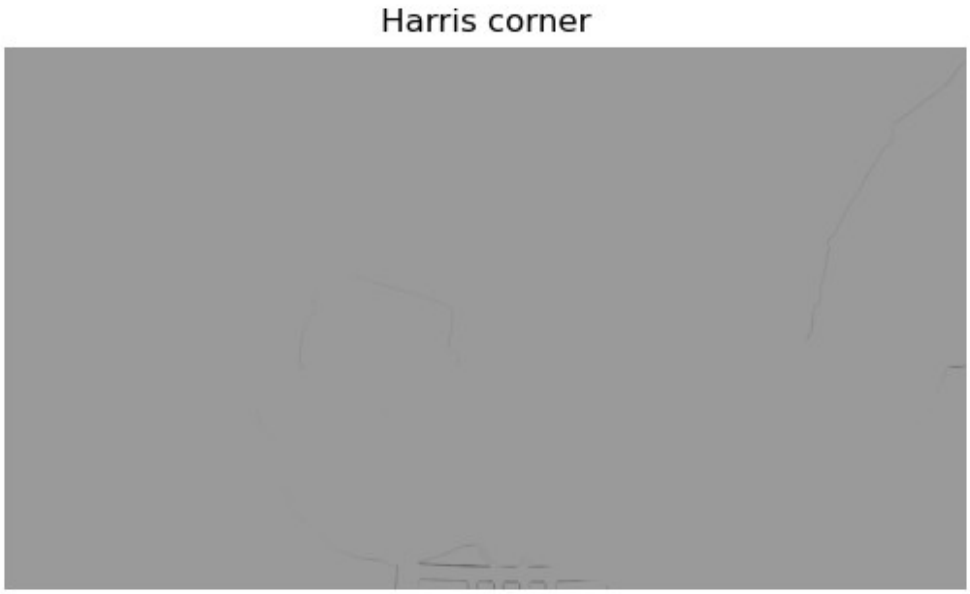

```
plt.imshow(image_orb, cmap='gray')
plt.title('ORB')
plt.axis('off')
plt.show()
```



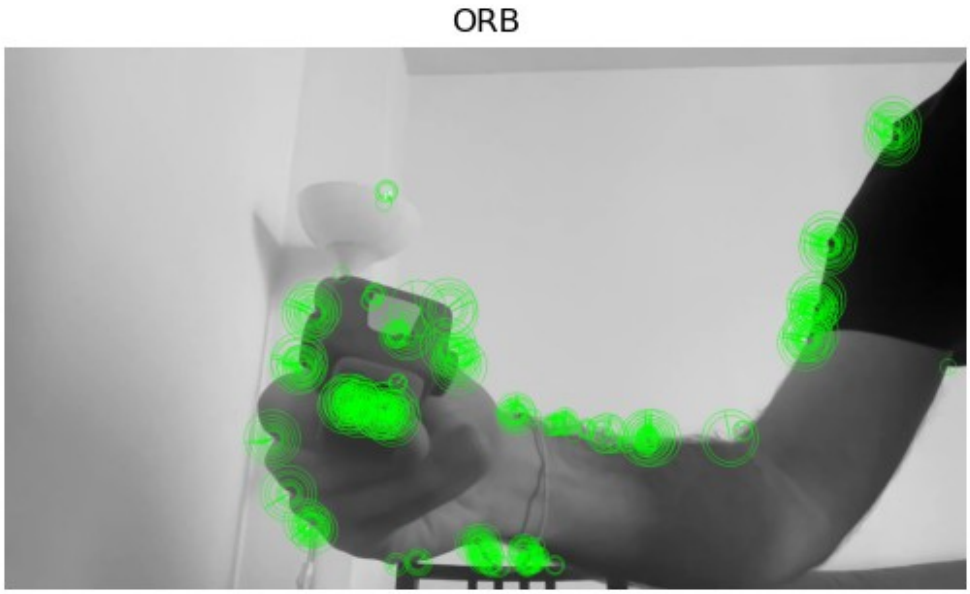

```
plt.imshow(image_sift, cmap='gray')
plt.title('SIFT')
plt.axis('off')
plt.show()
```

SIFT

```
harris_2 = cv2.cornerHarris(gray2_32, 2, 3, 0.04)

keypoints_sift_2 = sift.detect(gray2, None)
keypoints_orb_2 = orb.detect(gray2, None)

image_sift_2 = cv2.drawKeypoints(gray2.copy(), keypoints_sift_2, None,
color=(0,255,0), flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
image_orb_2 = cv2.drawKeypoints(gray2.copy(), keypoints_orb_2, None,
color=(0,255,0), flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

plt.imshow(harris_2, cmap='gray')
plt.title('Harris corner')
plt.axis('off')
plt.show()
```

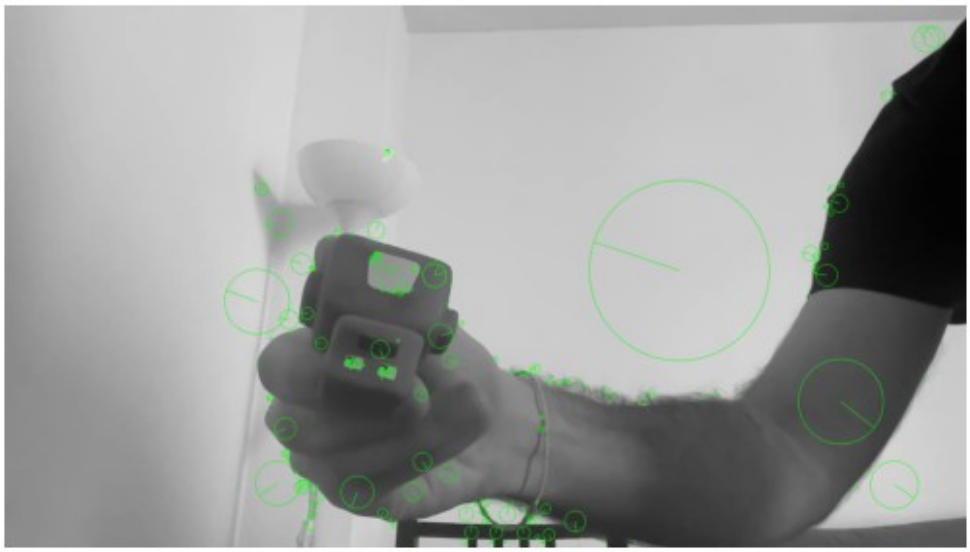



```
plt.imshow(image_orb_2, cmap='gray')
plt.title('ORB')
plt.axis('off')
plt.show()
```




```
plt.imshow(image_sift_2, cmap='gray')
plt.title('SIFT')
plt.axis('off')
plt.show()
```

SIFT



Harris corner detection didn't work as expected from the OpenCV tutorial. SIFT and ORB detected keypoints in the image, a lot of them are in the main object, both in the straight and rotated images.

For the tracking of the object I would say that both SIFT and ORB would work, first detecting keypoints and then matching them in subsequent frames thanks to descriptors. Both SIFT and ORB are invariant to the rotation and traslation (very important in this case since our object moves and rotates), SIFT is more robust on scale. Another thing to take in account is that ORB is way more faster and that can be important in tracking objects.