Visual Inspection of Motorcycle Connecting Rods

Students should develop a software system aimed at visual inspection of motorcycle connecting rods. The system should be able to analyze the dimensions of two different types of connecting rods to allow a vision-guided robot to pick and sort rods based on their type and dimensions. The two rod types are characterized by a different number of holes: Type A rods have one hole whilst Type B rods have two holes.

Task 1

Image characteristics
1. Images contain only connecting rods, which can be of both types and feature significantly diverse dimensions.
2. Connecting rods have been carefully placed within the inspection area so to appear well separated in images (i.e. they do not have any contact point).
3. Images have been taken by the backlighting technique so to render rods easily distinguishable (i.e. much darker) from background. However, for flexibility reasons the system should not require any change to work properly with lighting sources of different power.

Functional specifications
For each connecting rod appearing in the image, the vision system should provide the following information:
1. Type of rod (A or B).
2. Position and orientation (modulo π).
3. Length (L), Width (W), Width at the barycenter (WB).
4. For each hole, position of the centre and diameter size.

Task 2

While still meeting the requirement of the First Task, students should modify the system in order to deal with one (or more) of the following three changes in the characteristics of the working images:

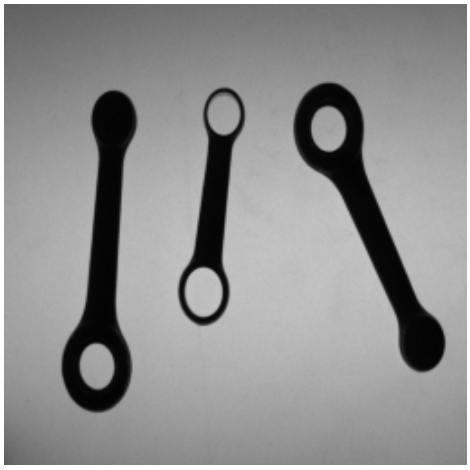
1. Images may contain other objects (i.e. screws and washers) that need not to be analysed by the system (such kind of objects are often referred to in computer vision as “distractors”).
2. Rods can have contact points but do not overlap one to another.
3. The inspection area may be dirty due to the presence of scattered iron powder.

Project Development

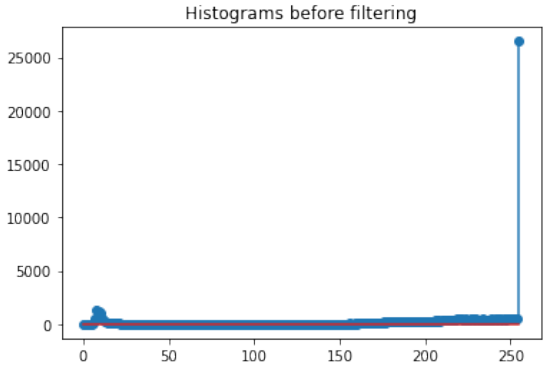
To solve the entire project I used Python and the library OpenCV in Jupyter Notebook environment. I report here only the important parts of the code and the related outputs, omitting some parts used for visualization purposes. The entire code with comments can be found in the file “Connecting Rods Project.ipynb”.

First Task

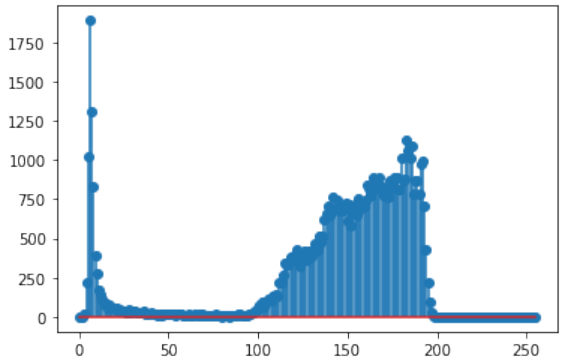
To perform the first task we have to binarize the image. To binarize the image we first look at the **graylevel histogram** to choose the best strategy to follow.
I focused on the first two images because they represent the same scene with different lighting conditions.



Img1



Graylevel Histogram of Img1



Graylevel Histogram of Img2

As we can see, in the first image there is more light and the background colour is not uniform.
In the second image the graylevels are differently distributed but both histograms can be tough of as **bimodal**.
First to all, a denoising filter could help binarization, so I applied a Gaussian Filter to both images, choosing the parameter σ as a tradeoff between denoising and blurring.

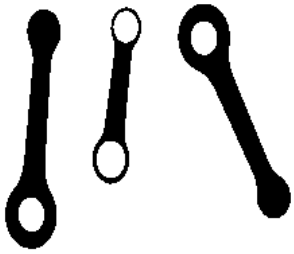
To perform binarization I tried some approaches. First of all I tried a naive implementation with static threshold.

```python
# Binarization "by hand" (without OpenCV functions)
# (Naive implementation)

def binarize(img, T):
    height = img.shape[0]
    width = img.shape[1]
    for i in range(height):
        for j in range(width):
            if img[i,j] <= T:
                img_bin[i,j] = 0 # black
            else:
                img_bin[i,j] = 255 # white
    return img_bin
```

As we could expect the result is not good and is not the same for the two images because they need a different threshold, being the light different. Using the cv2.threshold function I obtain the same results.
An adaptive method is not helpful in this case, however I tried it using the cv2.adaptiveThreshold function because I was curious. You can see the results uncommenting the right lines in the code.

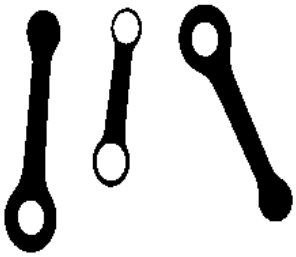
Being the histogram bimodal, we can apply the **Otsu**'s algorithm to find the right threshold independently from the light conditions. This can be implemented passing the right arguments to the cv2.threshold function.

Threshold = 128



Binary Img1

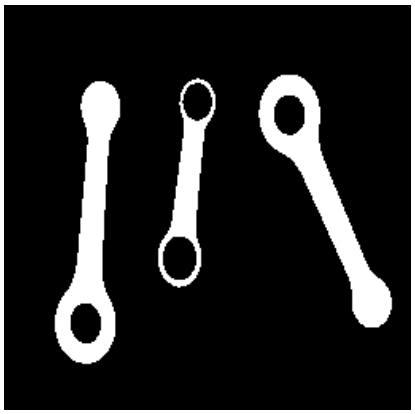
Threshold = 86



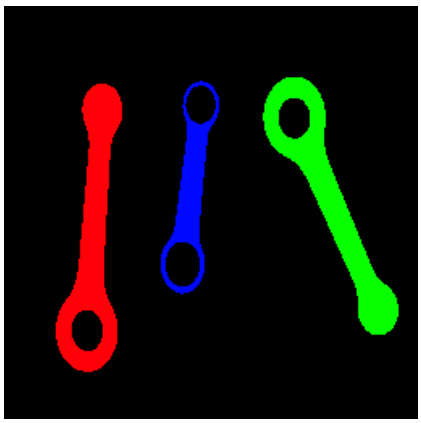
Binary Img2

As we could expect the result is good and is the same for the two images, so from now on I focus only on the first one.

Now we have to perform the **Connected Components Labeling** on the binary image. I used the cv2.connectedComponents function. Note that this function works with white components, so we have to invert the image to make it work properly.



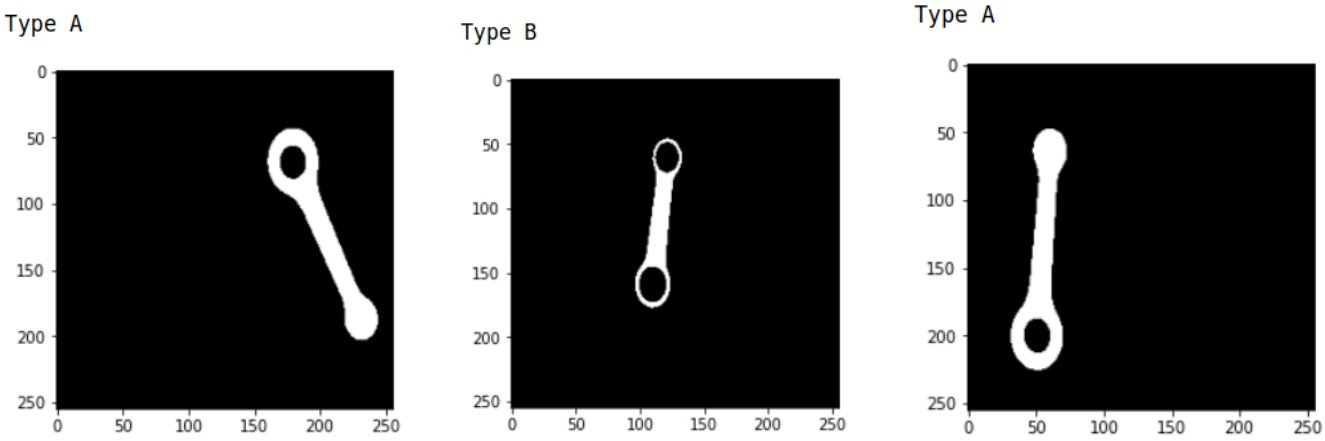
Inverted image



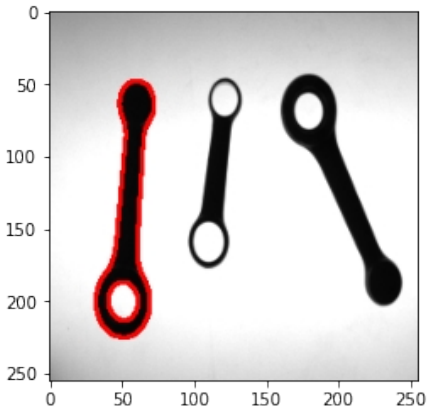
Labeled image

The function is also able to return the number of connected components, in this case 3.

To analyze each component independently I applied a mask on the inverted image. The following analysis are executed for every component.
Exploiting the cv2.findContours function we can understand the number of holes in the rods (and thus the type). In particular, if 2 contours are found (outer contour + hole contour) the rod is of type A, if 3 contours are found (outer contour + 2 holes contour) the rod is of type B. The output is the following.



To have a graphical idea of the found contours:



Inner and outer contours of the first object.

To compute the **barycenter** position we can use the formula $x = M_{10}/M_{00}$, $y = M_{01}/M_{00}$ (where M00 is the Area) . To know the **moments** we can use the apposite function of openCV that returns spatial moments, central moments and central normalized moments, but here we need only the spatial ones because we don't have to compare different pieces so we don't need rotation and scale invariancy. The following results refer to the rod on the left in the above image.

```
def get_barycenter_and_area(img):
    moments = cv2.moments(img) # returns a dictionary ('moment':value)
    # from the last link above we know that also the pixel intensity is multiplied to compute the moments,
    # but we don't care because this multiplication is performed for all the images under analysis and
    # always the same number considered (255)
    # coordinates of barycenter
    Bx = int(moments["m10"] / moments["m00"])
    By = int(moments["m01"] / moments["m00"])
    area = moments["m00"]
    return Bx,By,area,moments

Bx,By,area,moments = get_barycenter_and_area(mask)
print("Coordinates of barycenter: x = {}, y = {}. Area = {}" .format(Bx,By,area))
```

```
Coordinates of barycenter: x = 54, y = 141. Area = 832065.0
```

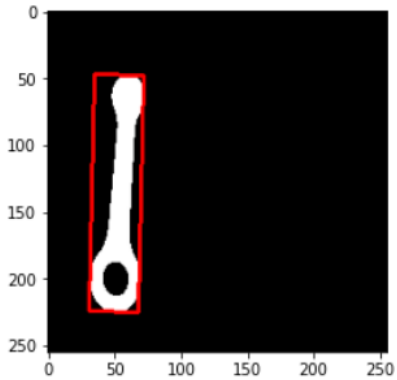
To compute the orientation with respect to the horizontal axis we can use the formula reported on the course slides, passing the central moments that we have from the previous function, or an openCV function like cv2.fitEllipse, that returns also the orientation angle that spaces from -90 to +90 degrees.

```
# The orientation is determined modulo Pi (from -90 to 90)
def get_orientation(moments):
    #teta_rad = -0.5*np.arctan((2*moments["mu11"])/(moments["mu02"]-moments["mu20"]))
    #teta = (np.degrees(teta_rad)) # abs to get the modulo pi angle
    (x,y),(MA,ma),teta = cv2.fitEllipse(contour[-1])
    # fitEllipse function returns angle wrt the vertical axis
    return (90 - teta)


teta = get_orientation(moments)
print("Orientation: teta = {:.1f}" .format(teta))
```

```
Orientation: teta = 86.1
```

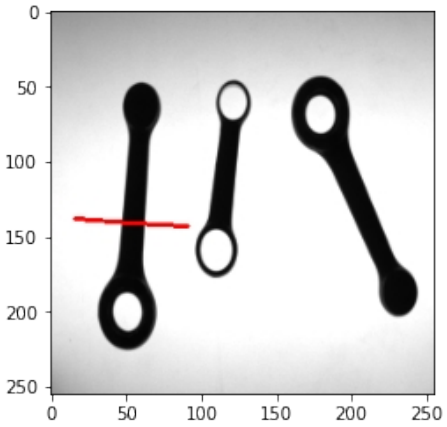
To find **Length** and **Width** we can use the Minimum Enclosing Rectangle (**MER**): the dimensions of the MER are the desired ones. The function cv2.minAreaRect, however, is not very precise, but I used it for simplicity. Another possible approach could be: use the fitEllipse function to compute the orientation, rotate the object to align his major axis to the vertical and use the cv2.minAreaRect function.
The following function compute L and W and draw the rectangle. L and W are expressed in pixel units.

```
def get_length_and_width(contour):
    MER = cv2.minAreaRect(contour[-1]) # -1 to get the outer contour and not the hole one
    MER_center = MER[0]
    MER_size = MER[1] # this is used, you can comment the others
    MER_angle = MER[2]
    box = cv2.boxPoints(MER)
    box = np.int0(box) # coordinates of the enclosing rectangle
    im = cv2.drawContours(mask_rect_rgb,[box],0,(255,0,0),2)
    plt.imshow(mask_rect_rgb)
    plt.show()
    return MER_size

MER_size = get_length_and_width(contour)
H = int(MER_size[0])
W = int(MER_size[1])
#print(MER_center,MER_size,MER_angle)
print("The piece has size: H = {}, W = {}" .format(H,W))
```




```
The piece has size: H = 177, W = 37
```

Width at the barycenter: we know the slope of a line and a point (barycenter). We need to compute the length of the line passing through the barycenter and with known angle with respect to the horizontal. Using the relations: $x_2 = x_1 + length\cos(\theta); y_2 = y_1 + length\sin(\theta)$ we can obtain 2 points and draw a line for visualization purposes using cv2.line(). Then, moving through this line we can initialize Wb as 0 and increment it if we encounter a point with intensity = 255 (foreground pixel). At the end we have **half** the width at the barycenter, so we can multiply by 2 or do the same computation in the opposite direction. Wb is expressed in pixels.
Note that the actually computation has some changed signs due to the downward y axis.
Another possible method could be to select the 2 points that belong to the outer contour and to the line and compute the euclidean distance between them.
Here are the results with the first approach.




```
# This function goes through the line and increment Wb if the pixel under analysis belongs to foreground
def get_Wb(Bx,By,teta):
    P = np.array([[Bx,By]],dtype=np.int32)
    P2 = np.zeros((1,2),dtype=np.int32) # initialize P2 as zeros
    # teta spaces from -90 to +90
    if (teta > 0):
        angle = np.radians(teta - 90)
    else:
        angle = np.radians(teta + 90)
    i = 0
    Wb = 0
    for i in range(int(length)):
        P2[0,0] = int(P[0,0] + i * np.cos(angle))
        P2[0,1] = int(P[0,1] - i * np.sin(angle)) # - because the axis is directed downward
        #print(P2)
        #print()
        if (mask[P2[0,1],P2[0,0]] == 255):
            Wb+=1
        i+=1
    return Wb

Wb = get_Wb(Bx,By,teta)
Wb *= 2
print("The width at the barycenter is: Wb =", Wb)
```

```
The width at the barycenter is: Wb = 20
```

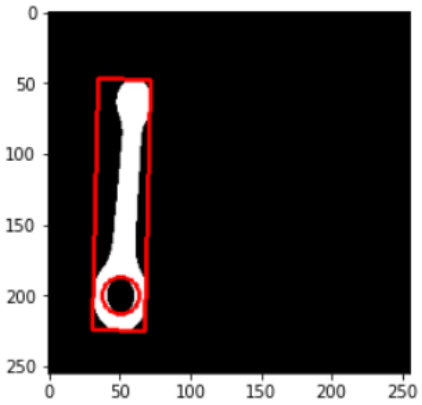
Now we have to find the **position of the centre** and the **diameter size** of each **hole**.
There are several ways to do that:

- **Hough Transform** for circle detection: it won't work in this case with openCV functions because the holes in the connecting rods images are ellipses.
- Use the **minimum enclosing circle** and its coordinates and radius for the hole(s) contour(s).

- Use the **generalized Hough Transform**.

The second method is the simplest one (but the less accurate).
Note that if the rod is of type B we have to find two circles.

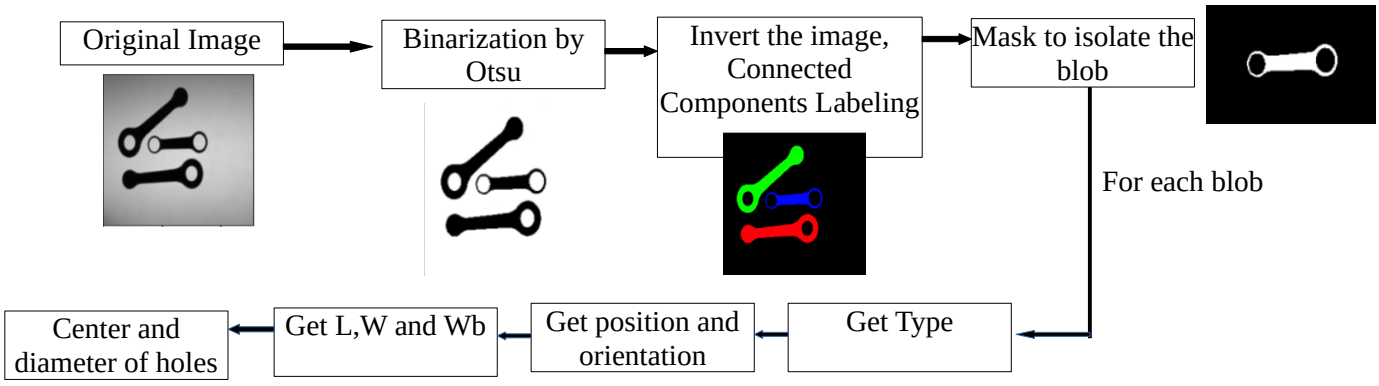
```
# Find center coordinates and radius using the Minimum Enclosing Circle applied on the inner contour
# N.B. IF THERE ARE 2 HOLES (TYPE B) WE HAVE TO FIND 2 CIRCLES!

def get_holes(contour,Type):
    (x,y),radius = cv2.minEnclosingCircle(contour[0])
    # cast to int to draw the circle
    x = int(x); y= int(y); radius = int(radius)
    cv2.circle(mask_rect_rgb, (x,y), radius, (255, 0, 0), 2)
    if (Type == 'B'):
        (x2,y2),radius2 = cv2.minEnclosingCircle(contour[1])
        # cast to int to draw the circle
        x2 = int(x2); y2= int(y2); radius2 = int(radius2)
        cv2.circle(mask_rect_rgb, (x2,y2), radius2, (255, 0, 0), 2)
        plt.imshow(mask_rect_rgb)
        plt.show()
        return x,y,radius,x2,y2,radius2
    # visualize the drawn circles
    plt.imshow(mask_rect_rgb)
    plt.show()
    return x,y,radius

if (Type == 'A'):
    x,y,radius = get_holes(contour,Type)
    print("The hole position is: x = {}, y = {}, the diameter is: D = {}" .format(x,y,2*radius))
else:
    x,y,radius,x2,y2,radius2 = get_holes(contour,Type)
    print("The holes positions are: x = {}, y = {} with diameter: D = {}; x2 = {}, y2 = {} with diameter2: D2 = {}"
```

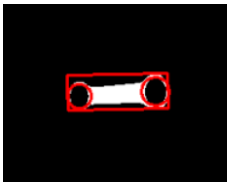



```
The hole position is: x = 51, y = 200, the diameter is: D = 26
```

Generalization of the procedure

This section in the code file contains the procedure to extract and analyze each blob of each image.
This was used to prove that what is written above is effective for all the images of Task 1.
To recap, the procedure followed for each image is as follows.





```
Type B

Coordinates of barycenter: x = 128, y = 116. Area = 376380.0
Orientation: teta = 3.6

The piece has size: H = 100, W = 35
The width at the barycenter is: Wb = 24

The holes positions are: x = 87, y = 119 with diameter: D = 24; x2 = 163, y2 = 114 with diameter2: D2 = 28
```

Second Task

I've chosen to deal with changes number 1 and 3.

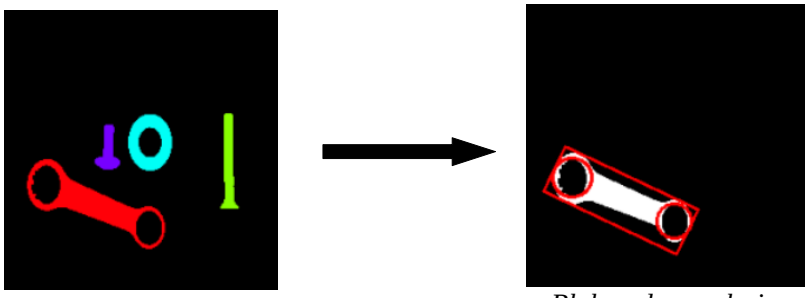
First change

To deal with other objects, we have to characterize the rods, e.g. in terms of Area, circularity, length, width...

The same initial analysis is performed (denoising, binarization, connented components labeling), then each blob is analyzed and if it has not the right characteristics it is discarded. By looking at the previous analysis results we can claim that the area of the smallest rods is about 370000, and processing the images of the second task we see that the biggest piece that have to be discarded has area about 270000. So we can filter out all blobs with area < threshold that can be chosen = 300000. For this reason, as soon as the blob is recognized, the area is computed and an if statement is used to discard the distractors.

The generalization of the procedure is the same of the first task except for the fact that the area is the first feature to be computed because is used in the condition of the if.

Some exemplar results are reported below.



Labeled image *Blob under analysis*

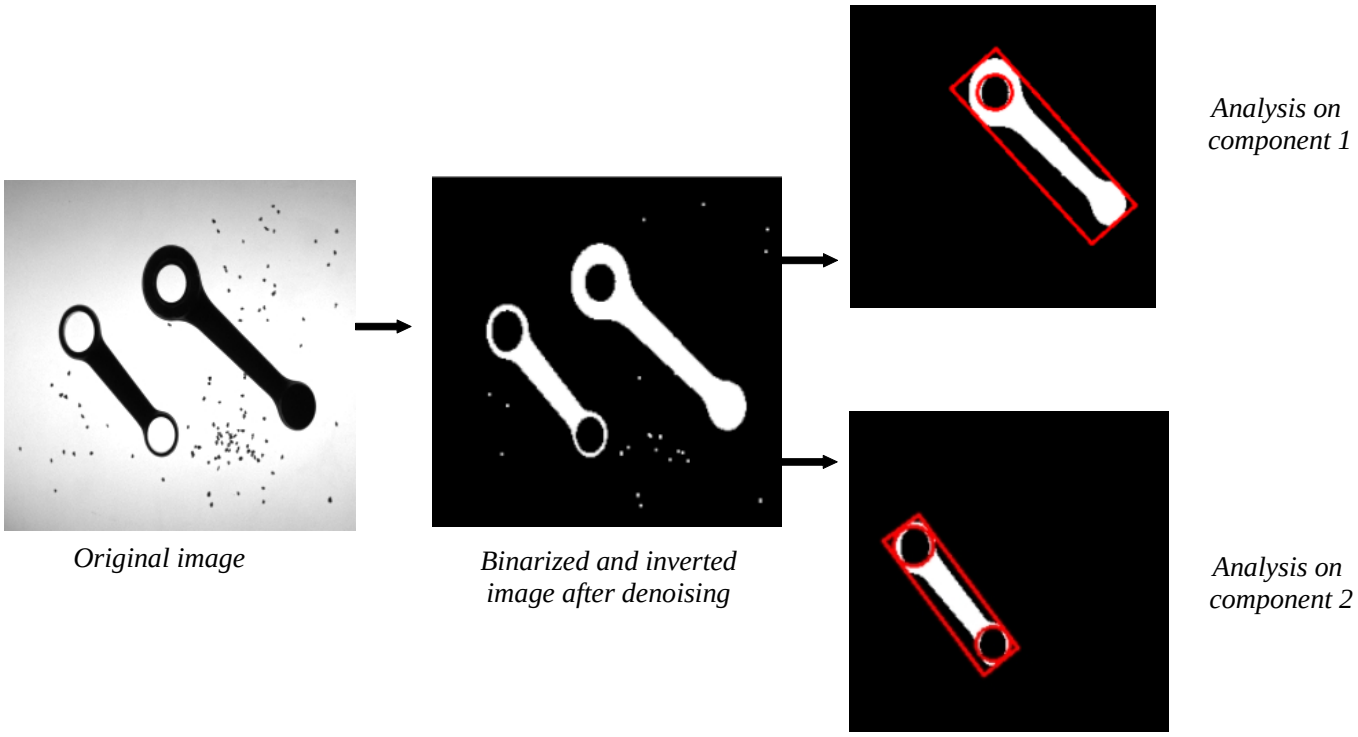
```
Coordinates of barycenter: x = 85, y = 175. Area = 596700.0
Orientation: teta = -22.6

The piece has size: H = 42, W = 128
The width at the barycenter is: Wb = 22

The holes positions are: x = 134, y = 196 with diameter: D = 30; x2 = 45, y2 = 158 with diameter2: D2 = 34
```

Third change
To eliminate the iron powder we can treat it as salt-and-pepper noise and use a **median filter**. I defined a new function to apply median and gaussian filters. However, since the powder grains can be big, this is not sufficient.
Hence I perform also an **opening operation**. The remaining powder grains will be exluded due to the area analysis highlighted in the first change. I used a 2x2 kernel because a bigger one would have damaged the rods with a small width around the holes.

So the differences from the First Change procedure are the different denoising function and the addition of the opening operation performed immediately after the inversion of the binary image.

```
# Apply median filter and gaussian filter (see Task 2, third change)
def double_denoise (img):
    img_filtered = cv2.medianBlur(img, 3)
    sigma = 0.5
    k_size = int(np.ceil((3*sigma))*2 + 1) #rule of thumb for a good kernel size given sigma (see slides)
    img_filtered = cv2.GaussianBlur(img, (k_size,k_size) , sigma)
    return img_filtered

def opening(img):
    kernel = np.ones((2,2),np.uint8)
    img_bin = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel)
    return img_bin
```

Here an exemplar result.



Original image

Binarized and inverted image after denoising

Analysis on component 1

Analysis on component 2