Alma Mater Studiorum - Università di Bologna

Ingegneria Informatica Magistrale

Computer Vision and Image Processing M

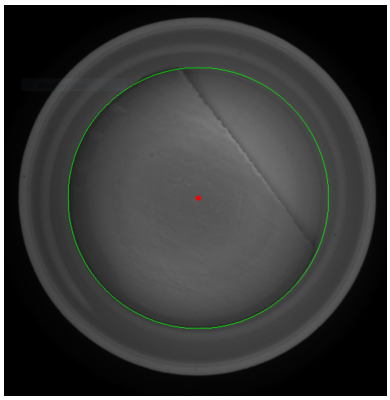
# Caps inspection

***Project 1 Plastic cap liner inspection***

***https://github.com/ilceltico/CV-cap-liner-inspection***

*Authors:*
Federico Domeniconi
Andrea Gavagna
Federico Stella

*Professor:*
Prof. Luigi Di Stefano



October 2019

# Contents

# 1 Introduction

## 1.1 Project requirements

The main task of the project is to inspect plastic caps in order to extract features like diameter and center, and to determine which of them have defects and which are good samples instead.

Each cap is composed of two parts: the plastic shell and the cardboard liner. The requirement is to develop a computer vision software that is able to outline the cap (figure 2), outline the liner (figure 3), if present, and record the position of the center and the diameter of the outer circle and the liner. For these computations, the software should make use of the Circle Hough Transform.
Moreover, the software must control if the liner is present or not (by checking the average lightness of the liner) and if it is complete or not (by computing the magnitude of the gradient to find internal edges, indicating a cut in the liner). In the latter case, it is also optionally required to outline the straight edge of the incomplete liner (figure 4).

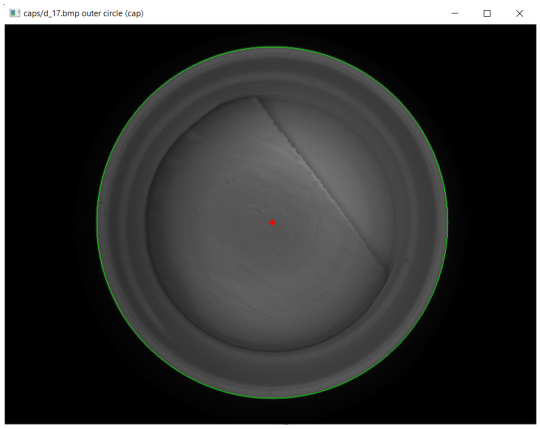



Figure 2: Outline the cap.

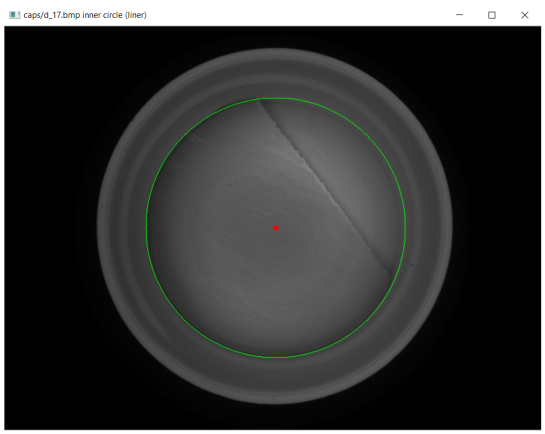



Figure 3: Outline the liner.

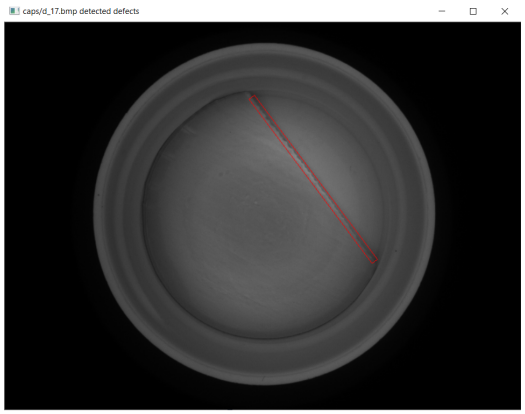



Figure 4: Outline defects.

As a requirement, project hints should be followed. The programming language can be chosen among C, C++ and Python, whereas the computer vision library to be used is OpenCV.

## 1.2 Report preface

We approached this project by following the hints, but we soon encountered some technical limitations and difficulties that led us to find alternative solutions and develop our own methods and algorithms. We tried a lot of different possibilities but, of course, only the best ones among them will be described in detail.
The description of our work will not directly follow the execution order of the code: it will mainly follow the chronological order of implementation, starting each task with naive ideas that will be gradually enriched, so as to make it easier to explain the motivations behind our choices. The alternative methods will be presented in detail when they are used for the first time, briefly stating pros and cons for each of them and the reason for their existence. The last section will be a comparative analysis of the precision and performance of the developed methods.

As for project technicalities, the chosen programming language is Python (+ Numpy), mainly because of its intrinsic quick prototyping abilities and easy syntax, especially useful with OpenCV's burdensome data structures and APIs. Program configuration, specifying the algorithm to use and some parameters, is handled with a separate JSON file.

## 1.3 Code structure

The repository contains a `/caps` folder, with test images for the code; a `/tests` folder, with older code, tests and test results; and a folder for organizational purposes.
Apart from these folders and the `README.md`, the root directory contains 4 Python files and a JSON file, for the configuration parameters. The Python files are:

- `program.py`, which contains the code for the main control flow and some internal functions for clarity and code reuse;
- `loadconfiguration.py`, which is the only module that needs to know the details of the configuration file. Its purpose is to load the file and initialize the global constants for execution;
- `utils.py`, which contains different useful functions that will be explained in detail in the next sections;
- `circledetection.py`, the richest file, in which all the non-library functions related to circle detection are contained;
- `performancetester.py`, which can be executed to asses the performances of the algorithms.

The code is thoroughly commented to ease comprehension, except for `performancetester.py`, leaving to this report the theoretical explanations.
A code documentation is included: each function is described in detail, indicating the use of parameters, the structure of return values and the presence of possible exceptions.

## 2 Outlining the cap

In order to outline the circular cap mouth one needs to generate a circle that fits it, and this can be done using the Circle Hough Transform. Considering $(x_c, y_c)$ as the center of the circle and $r$ as its radius, its mathematical representation is $(x - x_c)^2 + (y - y_c)^2 = r^2$. Each point of the image is mapped, by means of the aforementioned equation, to the so-called parameter space, which is 3-dimensional since it contains $x_c$, $y_c$ and $r$ coordinates. Such mappings produce a double cone for each input point $(x, y)$, which is centered on $(x, y, 0)$.
In order to compute the most likely circles from this space (i.e. the points of the parameter space with the most intersections), it is necessary to work with a discretization of it: a 3D accumulator array. Each point casts a vote to a cell of the array if and only if its double cone passes through it. The problem with this method is that the 3D array can get extremely big, slowing down the voting process and requiring lots of memory. For this reason different methods have been proposed in literature, with OpenCV embracing the *2-1 Hough Transform (21HT)*.

In a mathematical circle, the radii are orthogonal to the circumference points and the center lies on the intersection of the radii. Thus, the search for centers can be sped up by focusing on the normal direction with respect to the edge points, and this direction is represented by the gradient of such points.
As described in [4], the 21HT uses these principles and reduces the storage requirement by decomposing the problem in two stages: a first stage to find the center and a second stage to find the radius. It first computes the gradient direction of each edge point and, after this, each edge point votes for every point along its gradient direction, with these votes being accumulated in a 2D array. Then it takes the centers with more votes than a set threshold (`param2` in the OpenCV function) and, for each of them, it computes a histogram of the distances from the candidate center to the edges (i.e. the radii). For this reason, OpenCV's official documentation states that the function `HoughCircles()` detects the centers well, but it may fail to find the correct radius. In order to help the function, the user is allowed to specify a radius range (`minRadius` and `maxRadius` parameters). The documentation also suggests to take the center only, while computing the radius using an additional procedure.

An additional problem of the `HoughCircles()` function is that it includes

Canny's edge detection algorithm, prior to the execution of the 21HT, without letting the user specify all of its parameters. This leads to some problems that will be described and solved when they arise, but it already represents a limitation since it is impossible to let the function find the outer circle in an efficient and robust way. In fact, the embedded edge detection algorithm tends to find more points than necessary, reducing the efficiency and decreasing robustness, while the boundary between the cap and the black background appears to be trivially computable, for example through binarization, yielding cleaner results. Thus, our approach takes the path of binarization, as usual for CV tasks with black/white foreground against a white/black background.

## 2.1 Image binarization

In addition to being used for a robust execution of `HoughCircles()`, the binarization process will also help the subsequent tasks by providing a mask that perfectly matches the cap.
It is performed in two steps (code snippet 1):

1. Image thresholding using *Otsu*'s algorithm to find the optimal threshold. The grey-level histogram is quite clearly bimodal but some dark pixels are part of the liner border, inside the cap. Therefore, some images can result in a blob with small holes (figure 5).

2. Performing a closing operation with a 7x7 kernel, in order to close the aforementioned holes. This morphological operation has been chosen because it can fill the holes while preserving the overall shape of the cap (figure 6).


Figure 5: After Otsu's algorithm.

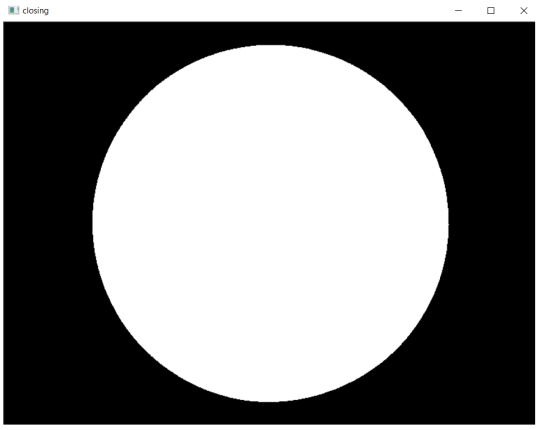



Figure 6: After the closing operation.

At this point, it is already possible to apply the 21HT to find the outer circle. Nevertheless, we decided to include a control for the circularity of the cap, because if it's not circular enough the results from the HT are not reliable and one may decide beforehand to discard the picture or retake it.

Code Snippet 1: `utils.binarize(img)` function, binarization of the image.

```
1  #Otsu's Thresholding
2  ret, thresh_otsu = cv2.threshold(img, 0, 255,
   ↪  cv2.THRESH_BINARY+cv2.THRESH_OTSU)
3
4  #Morphological Closing operation
5  kernel = np.ones((7,7), np.uint8)
6  closing = cv2.morphologyEx(thresh_otsu, cv2.MORPH_CLOSE, kernel)
```

## 2.2 Circularity measures

In the mathematical literature there are plenty of options for circularity measures (see [3] for examples), with *Compactness (Form Factor)* and *Haralick's Circularity* being two of the best known measures and more than enough for our purposes.

**Compactness** The form factor is defined as $C = P^2/A$, where P is the perimeter of the blob and A is its area. It is a dimensionless quantity and thus a scale invariant feature. In a continuous environment, it's clear that this measure is minimized by regular polygons, ultimately reaching its minimum with a circle ($4\pi$). In a discrete environment, though, this doesn't quite hold, because of the connectivity constraints for discrete perimeters and their computations.

**Haralick's Circularity** Haralick's Circularity overcomes these problems and is perfect for discrete environments. It is the ratio between the mean distance of the edge points from the barycentre ($\mu = \frac{1}{m}\sum_{k=1}^{m} d_k$, with $d_k$ distance of the k-pixel from the barycentre) and the standard deviation of such distances ($\sigma = \sqrt{\frac{1}{m}\sum_{k=1}^{m}(d_k - \mu)^2}$):

$$\widetilde{C} = \frac{\mu}{\sigma} \tag{1}$$

The smaller the standard deviation, the more similar the distances will be to one another, and thus the higher the circularity. As for the compactness, it is a dimensionless (i.e. scale invariant) feature.

The function `utils.haralick_circularity()` thus computes the Haralick's circularity of the binarized image of the cap to check if it is circular enough. The threshold has been set to 200, since all the images have a circularity that ranges from 243 to 657. This is consistent to what was found in [3], as images with circularity values spanning the $10^2$ order of magnitude are great circles.

For the actual computation of the Haralick's circularity one needs the contour pixels of the blob, and we identified four main strategies to get these pixels:

- dilation of the binarized image (3x3 square as a structuring element) followed by a subtraction, producing the outer contour of the shape;
- erosion of the binarized image (3x3 square as a structuring element) followed by a subtraction, producing the inner contour of the shape;
- use of the OpenCV's function `cv2.Canny()` (*Canny's edge detection)*
- use of the OpenCV's function `cv2.findContours()`.

We've chosen the latter, being both precise and the easiest one to use: thanks to binarization there is no need to use slower and more complex edge detection algorithms.
From the contour, the barycentre is computed by means of the `moments()` function, since the ratio between $M10$ (i.e. the sum of $i$ coordinates of each perimeter pixel) and $M00$ (i.e. the area) is the $i$ coordinate of the barycentre and the ratio between $M01$ (i.e. the sum of $j$ coordinates of each perimeter pixel) and $M00$ is the $j$ coordinate. At this point, mean distance and variance are trivially computed.

Code Snippet 2: `utils.haralick_circularity()` function.

```
1  def haralick_circularity(binary):
2      contours, hierarchy = cv2.findContours(binary, cv2.RETR_TREE,
       ↪  cv2.CHAIN_APPROX_NONE)
3      cnt = contours[0]
4      moments = cv2.moments(cnt)
5      i_b = int(moments["m10"] / moments["m00"])
6      j_b = int(moments["m01"] / moments["m00"])
7      diff = cnt - np.full_like(cnt,[i_b, j_b])
8      diff = (diff).reshape(len(diff),2)
9      distances_from_bary = np.linalg.norm(diff, axis=1)
10     average_distance = np.sum(distances_from_bary) /
       ↪  len(distances_from_bary)
11     variance = np.sum(np.square(distances_from_bary -
       ↪  average_distance)) / len(distances_from_bary)
12     haralick_circularity = average_distance / np.sqrt(variance)
```

## 2.3 Hough Transform

At this point, we are completely sure that the cap is a good-enough circle and we can apply a Hough Transform. Thanks to binarization, the embedded Canny's edge detection in OpenCV's 21HT works perfectly, even if the user can't specify all of its parameters, and it finds perimeter points only. The most-voted circle found by this function is already a decent result, but to achieve something

more precise one can get the best $n$ circles and average them out. This is exactly what happens by modifying the parameter `average_best_circles` in the configuration file, with the best results being produced by setting it to 3. The function `circledetection.find_circles_hough()` fully encapsulates the 21HT, adding the possibility to set this parameter.
Considering the radius problem of the 21HT, another interesting approach is to totally discard the radii provided by the function. By keeping a center only (averaging them out if there's more than one), one can compute the radius by averaging the distances between the center and the perimeter points, and this can be easily done thanks to binarization. This produces an average radius, instead of a mode, which is what the histogram would compute. Of course this approach assumes the previous operations to yield perfect results in order to correctly work, but this was easily achievable, at least in the test images, finding precisely and only the perimeter points. The parameter `radius_computation`[1] can thus be set to `mean_radius`, in order to compute the radius as a mean of the radii of the best circles found by the 21HT (figure 7), or to `border_distance`, in order to perform the aforementioned operation (figure 8).

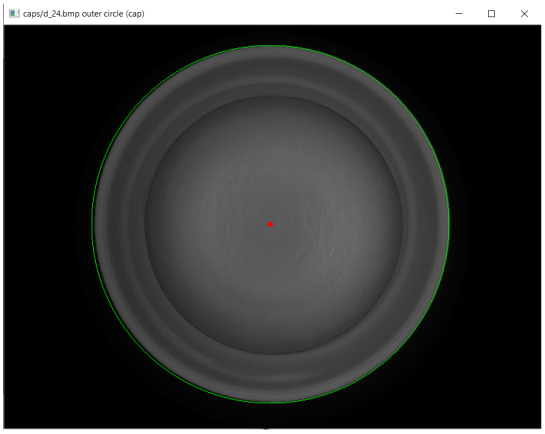



Figure 7: Pure `HoughCircles()`, cap 24.

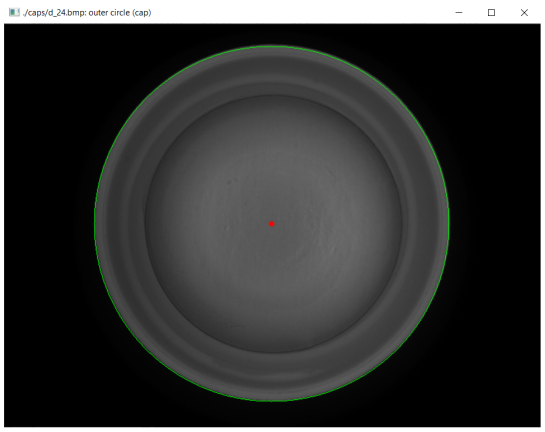



Figure 8: Manually computed radius.

Code Snippet 3: `program.outer_circle_detection()` function, Hough Transform.

[1] In the whole report we refer to radius computation, but of course the program will display a diameter as specified in the requirements.

```
    # Compute the HT and average out the best circles, as many as specified in the config
    x, y, r = circledetection.find_circles_hough(img, 1, 1, 200, 5, 0, 0, config.OUTER_HOUGH_NUMBER_AVG)

    if config.OUTER_RADIUS_COMPUTATION == 'border_distance':
        # Compute the radius as the mean distance between points and the previously found center
        edges = cv2.Canny(img, 100, 200, L2gradient=True)
        pixels_y, pixels_x = np.nonzero(edges)

        r = sum(np.sqrt((pixels_x - x)**2 + (pixels_y - y)**2)) / len(pixels_x)
```

As for parameter tuning, in order, a resolution factor 1 achieves the required precision while still being fast enough, and enables us to let the function find multiple nearby centers, to average them out, and this also explains the "minimum distance" parameter set to 1. Canny's high threshold has been set to 200, while the accumulator threshold to accept a center has been set to 5, a low number, to be able to always get enough circles for the subsequent mean computation. Minimum and maximum radii have been set to 0, which lets the function ignore the parameters, since the program has no clue about how big the circle can be at this point[2].

The best configuration options are `average_best_circles = 3` and `radius_computation = "border_distance"`.

## 2.4 Least Squares Linear Regression

Since we were able to nicely extract the perimeter points of the outer circle, we thought that an alternative way to compute the circle could have been to apply a form of Linear Regression to them. While the Hough Transform can be seen as an

[2]One could count the perimeter points in order to estimate a radius range, but we thought that this approach would have required too many assumptions.

estimator of the best circle by means of a mode, regression uses an average, so the approach is different but the results might be interesting. For our purposes, we chose an Ordinary Least Squares multiple regression method that will be described in this section, with the model not being multivariate: there will be multiple regressors, but a single regressand.

In a normal Least Squares Linear Regression framework, there is a vector of $k$ independent regressors $\mathbf{x}$ [3], whose relationship with the regressand $y$ is assumed to be linear. For the $i$-th measurement, with $\mathbf{x_i}$ as the vector of regressors:

$$y_i = w_0 + w_1 x_{i1} + w_2 x_{i2} + \cdots + w_k x_{ik}$$

In matrix form, having $\mathbf{w}$ as the vector of weights and re-defining $\mathbf{x_i}^T$ to be $[1 \quad \mathbf{x_i}^T]$ in order to avoid adding $w_0$:

$$y_i = \mathbf{x_i}^T \mathbf{w}$$

Letting $\mathbf{y}$ be the column vector of the different measurements for $y$, and $\mathbf{X}$ be the matrix of the different measurements for $\mathbf{x}$:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ y_n \end{bmatrix} \quad \mathbf{X} = \begin{bmatrix} \mathbf{x_1}^T \\ \mathbf{x_2}^T \\ \cdots \\ \mathbf{x_n}^T \end{bmatrix}$$

We then have:

$$\mathbf{y} = \mathbf{X}\mathbf{w}$$

Since this system is typically overdetermined, we can approximate a solution by minimizing a certain error function. Letting $E(\mathbf{w})$ be the sum of squared errors for predictions obtained with weight vector $\mathbf{w}$, we have:

$$E(\mathbf{w}) = \sum_{i=1}^{n} (\mathbf{x_i}^T \mathbf{w} - y_i)^2 = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$$

The gradient and Hessian are:

$$\nabla E(\mathbf{w}) = 2\mathbf{X}^T(\mathbf{X}\mathbf{w} - \mathbf{y})$$

$$\mathbf{H}(\mathbf{w}) = 2\mathbf{X}^T\mathbf{X}$$

[3] Non-transpose vectors are intended as column matrices.

One can find the minimum of the error function $E(\mathbf{w})$ by imposing its gradient to be 0, provided that the Hessian is positive definite. This condition holds, as the the Hessian turned out to be the product of a transpose matrix by the original one, whose columns represent independent variables. For this reason, the error function is convex and thus has a single minimum:

$$\nabla E(\mathbf{w}) = 2\mathbf{X}^T(\mathbf{X}\mathbf{w} - \mathbf{y}) = 0$$

$$\mathbf{X}^T\mathbf{X}\mathbf{w} = \mathbf{X}^T\mathbf{y}$$

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

With $\mathbf{X}^\dagger = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$ being the pseudo-inverse matrix of $\mathbf{X}$.

In order to apply this method to our points, we need to find a regressand which can be expressed as a linear combination of the regressors. Let's start from the equation of the circle, with $r$ as the radius and $(x_c, y_c)$ as the center:

$$(x - x_c)^2 + (y - y_c)^2 = r^2$$

$$x^2 + x_c^2 - 2xx_c + y^2 + y_c^2 - 2yy_c = r^2$$

The center and the radius are the parameters of this equation, representing a similar role to the weights $\mathbf{w}$. Moreover, $x$ and $y$ represent the data we have, and can carry out the function of the regressors and/or of the regressand. In order to find a linear function, we can define other variables to use, and these can also be non-linear functions of the original variables, as long as the resulting equation is linear. Since we need to have the parameters of the equation on one side only, a good result is the following:

$$2xx_c + 2yy_c + r^2 - x_c^2 - y_c^2 = x^2 + y^2$$

By defining the weight vector as $[x_c \quad y_c \quad r^2 - x_c^2 - y_c^2]^T$, we end up with the following matrix form, which constitutes a linear system:

$$\begin{bmatrix} 2x_1 & 2y_1 & 1 \\ 2x_2 & 2y_2 & 1 \\ \cdots & & \\ 2x_n & 2y_n & 1 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ r^2 - x_c^2 - y_c^2 \end{bmatrix} = \begin{bmatrix} x_1^2 + y_1^2 \\ x_2^2 + y_2^2 \\ \cdots \\ x_n^2 + y_n^2 \end{bmatrix}$$

This system can be solved in a least squares sense with the aforementioned method.

In order to speed up the computation, which involves matrix multiplication and inversion, we deployed a further analysis based on a short paper by R. Bullock [1]. Let $(u, v)$ be point coordinates in the reference system of the barycentre of all the points, with $(u_c, v_c)$ being the center of the circle; let $S_u = \sum_i (u_i)$, $S_{uu} = \sum_i u_i^2$ and so on; it is possible to find $(u_c, v_c)$ by solving the following system of equations:

$$\begin{cases} u_c S_{uu} + v_c S_{uv} = \frac{1}{2}(S_{uuu} + S_{uvv}) \\ u_c S_{uv} + v_c S_{vv} = \frac{1}{2}(S_{vvv} + S_{vuu}) \end{cases}$$

Which leads to the following formulas:

$$v_c = \frac{\frac{(S_{uuu}+S_{uvv})S_{uv}}{2} - \frac{(S_{vvv}+S_{vuu})S_{uu}}{2}}{S_{uv}S_{uv} - S_{vv}S_{uu}}$$

$$u_c = \frac{\frac{S_{vvv}+S_{vuu}}{2} - S_{vv}v_c}{S_{uv}}$$

With the radius being computed as:

$$r = \sqrt{{u_c}^2 + {v_c}^2 + \frac{S_{uu} + S_{vv}}{N}}$$

Computing the normal $(x_c, y_c)$ coordinates is now trivial.

These formulas are directly implemented in the function `fast_ols_circle_fit()` in `circledetection.py` and led us to the same results as the traditional pseudo-inverse computation but in a visibly shorter amount of time. Of course, this whole regression method assumes that the user can perfectly isolate the perimeter points, as even a single outlier could lead to bad results: it was possible to do this with the test images and it seems likely to be achievable in general for the outer circle, but of course this assumption makes this method less robust than the Hough Transform.
Precise performances are discussed in section 6.

Practically speaking, the circle detection for the cap is handled by `outer_circle_detection()`, which chooses the right method based on configuration parameters. If the user sets `method` to `least_squares` in the configuration file, it executes the function `find_circle_ols()` in `circledetection.py`, whose behavior can be set through the arguments that in turn depend on the configuration

parameters. Most of the parameters and possible behaviors of such function can be set for the inner circle detection only, and will be discussed later, since there is no need of them for the outer circle detection.
It's worth noting that this function, as a first task, labels all the edge points dividing them into separate connected blobs, that are then treated independently. This is done with the function `utils.get_blobs()`, which makes use of OpenCV's `connectedComponentsWithAlgorithm()` with the BBDT algorithm[2]. For this reason each blob generates a circle, and in order to generate a final circle it's possible to compute it as weighted mean of the circles found for each blob (with the weight being the number of points in that blob), by setting the configuration parameter `circle_generation` to `mean`, or by merging all the blobs and fitting everything again, setting it to `least_squares`. If the latter is selected, the regression is unnecessarily executed twice: once for each blob, and once for the merged blob, for reasons that depend on the function being used for inner circle detection too, where this double execution is needed. Of course this is wasting some computation time for the outer circle, but it allows us to nicely factorize all the regression methods into a single function, improving software elegance, and the regression method will turn out to be so fast that this is utterly irrelevant.

# 3 Is the liner missing?

By observing the test images, it is clear that the presence of the liner produces darker pixels in correspondence of the liner itself, which leads to a darker average lightness, and this is also the project hint to be followed (figure 9 and 10).
Since we have a mask that fits the cap, we are able to compute the average lightness of cap pixels only, basically achieving a sort of scale invariance. In fact, if we computed the average over the whole image we could answer to the missing liner question anyway, but the process wouldn't be robust to changes in the dimensions of the caps, as smaller caps would produce darker averages and vice versa.

To easily set a threshold for the average lightness, the function `get_missing_liner_threshold()` in `utils.py` is executed beforehand and automatically cycles over all the available images, computes the binarization and the mask, separately computes the average lightness of images with full liner and without the liner, and finally sets the threshold as the mid value between the two. This function simulates a learning algorithm that analyzes training images to compute a threshold.

If the average lightness of an image is higher than the threshold, the liner is missing, and as a consequence the following sections are not executed.

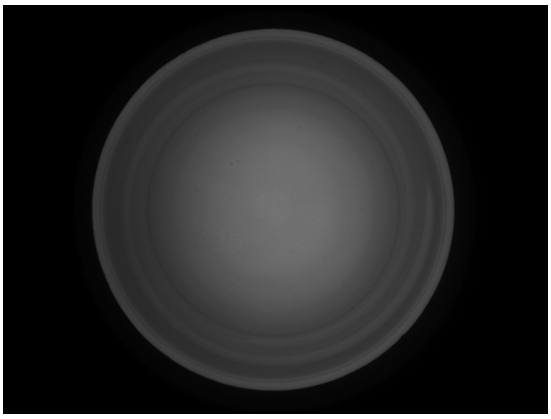


Figure 9: Cap 31, missing liner.

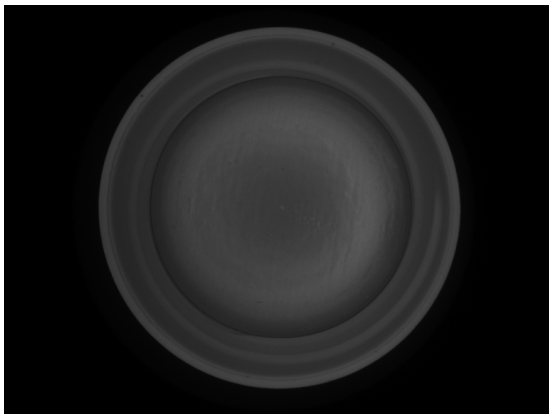


Figure 10: Cap 01, full liner.

# 4 Outlining the liner

The Circle Hough Transform is advised for this task too, but this time the limitations of the 21HT OpenCV implementation get more evident and will lead us to find some different solutions.

## 4.1 Linear contrast stretching

Nevertheless, the images appear lacking some contrast (figure 11) to be able to reliably isolate the perimeter of the liner, as the histogram spans a small grey-level range. Thus, a contrast enhancing operation is required to proceed anyhow: linear stretching, exponential operator and histogram equalization are three ways to have a better contrast, with also the possibility to set histogram percentiles for $r_{min}$ and $r_{max}$ in linear stretching.
The best results were obtained with a simple linear stretching (figure 12), and it's worth pointing out that it is performed on cap pixels only, by means of the mask. In this way, there's no need to manually set percentiles, as the background pixels that would be around 0 are excluded from the computation.

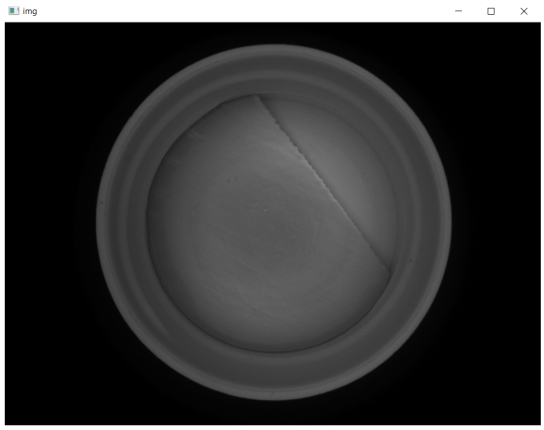



Figure 11: Original image.

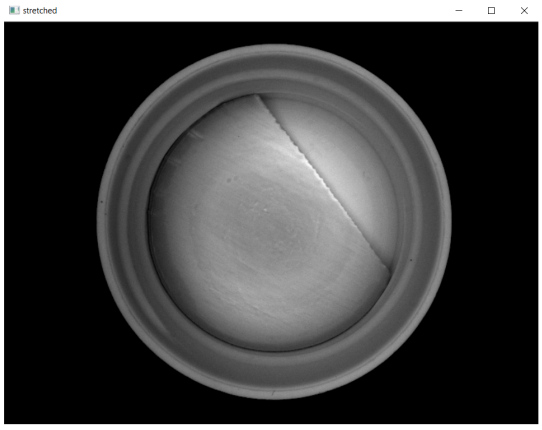



Figure 12: Linear stretching result.

Code Snippet 4: `program.main()` function, linear contrast stretching.

```
1  binary = utils.binarize(img)
2  mask = binary.copy().astype(bool)
```

```
3  stretched = ((255 / (img[mask].max() -
   ↪  img[mask].min()))*(img.astype(np.float)-img[mask].min())).astype(np.uint8)
4  stretched[~mask] = 0
```

## 4.2 Hough Transform

Directly applying the `HoughCircles()` function to the stretched image doesn't produce good results, as too many edges pop out: the cap, the liner, and also the thread of the cap. In order to smoothen and clean out the image a Gaussian filter is applied, with kernel size and $\sigma$ that depend on the particular method that is applied afterwards. At this point the function works better, but the aforementioned problem with the internal Canny's edge detection arises.

**Canny's edge detection.** The OpenCV function `Canny()` works as follows, according to the official documentation:

1. Noise reduction by means of a 5x5 Gaussian filter;
2. Computation of first partial derivatives ($I_x$ and $I_y$) with a Sobel operator, whose dimension can be set with the `aperture_size` parameter and has a default of 3;
3. Gradient computation. The magnitude formula can be selected with the parameter `L2gradient`: if `True`, it uses the L2 norm $G = \sqrt{I_x^2 + I_y^2}$, else it uses the L1 norm $G = |I_x| + |I_y|$. It's `False` by default;
4. Non-Maxima Suppression (NMS): checks the neighborhood of each pixel to compute whether that pixel is a local maximum along the gradient direction or not. This allows to find thin, single-pixel edges;
5. Hysteresis thresholding: a pixel is taken as an edge if its gradient magnitude is higher than $T_h$ (the higher threshold, parameter `threshold2`), or if it's higher than $T_l$ (the lower threshold, parameter `threshold1`) and it's also in the neighborhood of a pixel that has already been labelled as an edge.

`HoughCircles()` embeds this function, but doesn't expose all of its parameters: in fact, the only parameter available to the user is Canny's higher threshold. The

lower threshold is set as half of it, the Sobel size to 3 and the magnitude computation to the L1 formula. The problem arises especially because of the L1 formula: it computes larger gradients along diagonal directions, producing spurious edge points (figure 13 vs figure 14, computed with the same parameters). This wasn't a problem for the outer circle because the edge was very clean, but it can be a problem now.

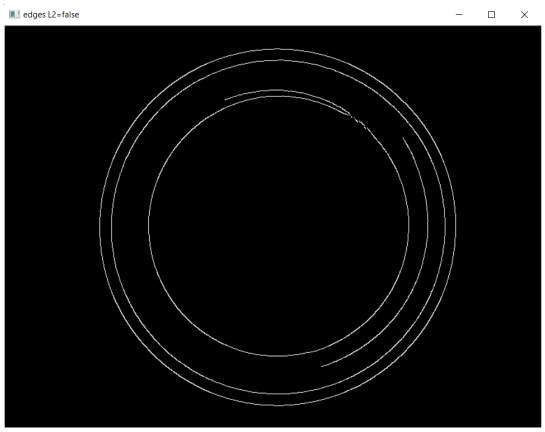



Figure 13: Edges with L1 formula.

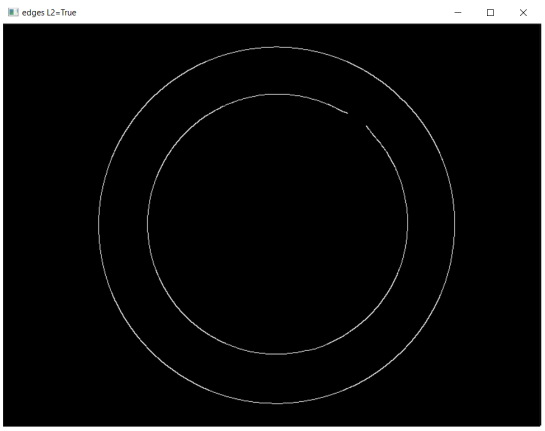



Figure 14: Edges with L2 formula.

Two solutions were found and implemented. The first one involves applying a 9x9, $\sigma = 2$ Gaussian filter prior to the execution of `HoughCircles()`. The second one involves a 7x7, $\sigma = 2$ Gaussian filter, followed by an explicit Canny's edge detection prior to `HoughCircles()`, whose embedded edge detection will find 2 edges for each edge resulting from `Canny()`, 1 pixel apart from each other. In this way we can set all the parameters of `Canny()`.
It's clear that the second method loses some efficiency, as the edge detection is executed twice, but the results produced by it are generally more precise (figure 15 vs 16). Moreover, it allows us to filter the results of the edge detection in order to rule out the outer circle of the cap by means of a circular mask[4], thus speeding up the voting process of the 21HT and ending up with a method that is actually faster than the straightforward HT. In addition, both methods set the maximum radius parameter of `HoughCircles()` to 0.98 or 0.90 times the radius of the outer circle. In this way it's possible to rule out bad circles without impacting the scale

[4]Implemented in `utils.py` as `circular_mask()`

invariance, since the outer circle is found scale-invariantly in each image.
The parameter `canny_precision` lets the user choose between `precise` (explicit `Canny()` with L2 gradient computation, figure 16) and `normal` (straightforward `HoughCircles()`, figure 15). The parameter `average_best_circles` can be set for the inner circle detection as well, with the same meaning as before. The best configuration consists in setting Canny's computation to L2 distance and choosing 2 as the number of circles to average.

As for the radius computation, using the same method of the outer circle seems impractical, as one should firstly isolate the points of the inner circle (excluding the outer circle and the liner defects). This operation can be done, for example, by means of 2 concentric circular masks with slightly different radii, but the results aren't as good as for the outer circle and the process seems a bit overcomplicated and arbitrary. Moreover, the radii computed by averaging the best circles found by 21HT are already good enough.

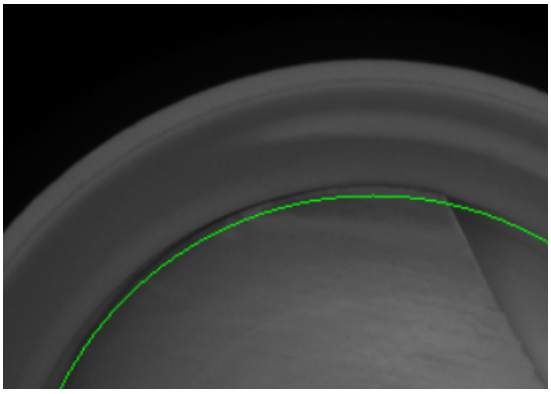


Figure 15: Cap 18, HT, Canny with L1 norm.

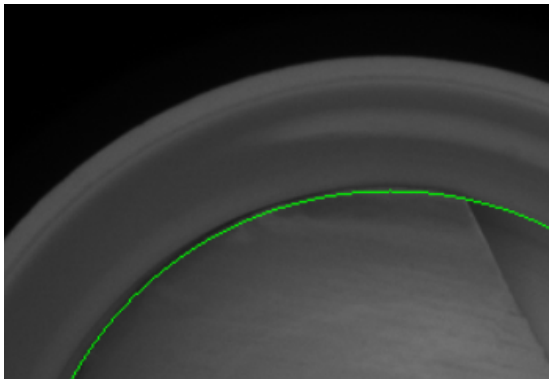


Figure 16: Cap 18, HT, Canny with L2 norm.

Code Snippet 5: `program.inner_circle_detection()` function, Hough Transform.

```
1  if config.INNER_CANNY_PRECISION == 'precise':
2          gaussian = cv2.GaussianBlur(img, (7,7), 2, 2)
3          edges = cv2.Canny(gaussian, 45, 100, apertureSize=3,
           ↪  L2gradient=True)
```

```
4
5           mask = utils.circular_mask(gaussian.shape[0],
            ↪  gaussian.shape[1], (outer_xc, outer_yc), 0.98*outer_r)
6           edges[~mask] = 0
7
8           x, y, r = circledetection.find_circles_hough(edges, 1, 1,
            ↪  100, 10, 0, np.round(0.98*outer_r).astype("int"),
            ↪  config.INNER_HOUGH_NUMBER_AVG)
9
10      else:
11          gaussian = cv2.GaussianBlur(img, (9,9), 2, 2)
12
13          x, y, r, = circledetection.find_circles_hough(gaussian, 1,
            ↪  1, 100, 10, 0, np.round(0.9*outer_r).astype("int"),
            ↪  config.INNER_HOUGH_NUMBER_AVG)
```

## 4.3 Least Squares Linear Regression

Analogously to what has been done for the outer circle, it seems a good idea to also apply OLS to this task. The problem, though, is that it's not possible to perfectly isolate the perimeter points of the inner circle, mainly because of the defects in the liner and the thread of cap, while the points of the outer circle are easily excluded by means of the already mentioned circular mask. As a consequence, in order to robustly and meaningfully apply the regression, it is necessary to develop some methods to eliminate the outliers.

### 4.3.1 Outliers

Outliers can be distinguished in 2 forms in the scope of this project: **blob outliers** and **pixel outliers**. We're naming "blob outlier" a blob, intended as a set of connected edge pixels, that is included in the regression but should not be: for example, the blob caused by the defect in the liner or by the thread is an outlier. On the contrary, a "pixel outlier" is simply an edge pixel that, even belonging to a blob that is not considered an outlier, is itself an outlier for that blob. This can

happen, for example, when the thread of the cap is connected to the inner circle, thus producing a single blob: the edge pixels belonging to the thread should be discarded from the regression, as they are outliers.
The next sections will thus describe: 2 methods to eliminate blob outliers; a method to partially solve the problem of semantically different blobs that get connected together because of proximity, thus enabling us to partially solve this problem with blob outliers elimination; a method to eliminate the remaining pixel outliers.

### 4.3.2 Blob outliers elimination

**Mean.** A first and quite naive method to eliminate blob outliers can be derived from the assumption that blob outliers weigh less than the sum of proper blobs in terms of number of pixels. Thus, if one uses regression to find a circle for each blob and then computes a weighted mean, one can eliminate the blobs that produced circles too far away from this mean. This is what happens in the function `outliers_elimination_mean()` in `circledetection.py`, that also allows to set distance thresholds for the acceptance of circles: the first number thresholds the distance of the centers, whereas the second thresholds the difference in radii.
As already stated, this method is quite naive, but it's very quick and works fine if the assumptions are satisfied. If more outliers are included, or if the outliers are very far away from the proper circle, it can discard both the outliers and the proper circle.
This function can be used by setting `outliers_elimination_type` to `mean`, and it returns the blobs that survived the elimination process, letting the user use them for further computation.

**Voting process.** This method is inspired from the Hough Transform itself: it produces a 3D (x,y,r) voting grid with the same shape of the image, and the radius ranging in [0, $max$(image height, image width)]. The actual number of pixels inside of this grid can be set by means of a resolution factor, that works exactly as the `dp` parameter of `HoughCircles()` (i.e. high value means low resolution). Each blob will then vote for the point in the grid that corresponds to its circle, with as many votes as pixels in the blob, and the point with most votes (i.e. the mode) wins. This point is not simply the winner tuple of parameters that will be used, as

for the HT: instead, it memorizes the blobs that voted for it. Thus, the function returns all the blobs that voted for the winning point, allowing the user to merge them together later for further computation.
The resolution factor is very important to this method, as it determines how close valid circles can be to each other: a resolution factor of 1 means that there are as many voting bins as pixels in the image, thus separating circle centers that are just 1 pixel away from each other; whereas a too high factor produces very imprecise results, failing to eliminate the outliers. The resolution factor that works best for us ranges from 8 to 64, that means that the voting grid has a resolution which is 8 to 64 times lower than the original image.
The actual implementation of this function is `outliers_elimination_votes()`, and it doesn't use a 3D structure, since it's too heavy. It uses a dictionary instead, with entries being generated only if a blob votes for them. In fact, in the 21HT a single point can vote for all the points along its gradient direction, thus making the dictionary approach less efficient (basically all the entries will be created sooner or later), while in this algorithm a blob votes for one point only[5]. This function can be used by setting `outliers_elimination_type` to `votes`.

### 4.3.3 Blob splitting

The two aforementioned procedures can be very useful, with the first one being effective especially if the blob outliers are small in number and size, and the second one being very effective if there is a multitude of blobs.
Having said this, the problem we have is that the defect of the liner and/or the thread of the cap could touch the inner circle, thus producing a single blob and escaping the blob outliers elimination. In order to partially deal with this, the parameter `split_blobs` lets the user choose to split all the blobs, producing many smaller blobs, by setting it to `true`. Moreover, the parameter `min_blob_dim` is numeric, and allows the user to specify also the minimum dimension for blobs: each blob will be split in as many blobs as possible, none of them being smaller than the specified size.
Of course, this procedure doesn't ensure the split to have a semantic meaning, but

[5]We also implemented the 3D structure, letting it execute only if the number of blobs exceeds a certain amount. This is probably unnecessary.

splitting the blobs helps, for example, the voting process of the previous section. Splitting blobs with 200 as minimum size followed by the voting process turned out to be a great solution, producing very good circles in very small amounts of time. To take into account the imprecise circles fitted on the small blobs, the resolution factor for the voting process is set to 64 (more on this in section 4.3.5)

### 4.3.4 Pixel outliers elimination

At this point, a set of blobs, each with its corresponding fitted circle, will result from the outliers elimination methods. In order to generate a final circle, it's possible to set `circle_generation` to `mean` or to `least_squares`, as for the outer circle. However, splitting the blobs doesn't completely terminate the job: in fact there is still the possibility that a small blob, which contains some pixels from the inner circle and some pixels, say, from the thread, isn't eliminated by the voting process. The amount of error in the inner circle at this point is usually close to zero, but some pathological cases may arise: caps `g_01` and `g_06` are two examples of what was just described (figure 17). The circle found by OLS is not bad at all, but it seems to be slightly shifted towards the part of the thread that is connected to the inner circle (figure 18). Blob outliers elimination helps a lot, and already provides a really good result, but we want to see if it's possible to fine tune the method by also eliminating some pixel outliers.

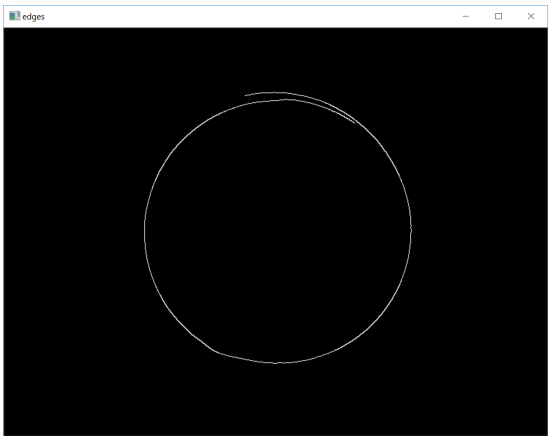



Figure 17: Cap 06, inner edges found with Canny.

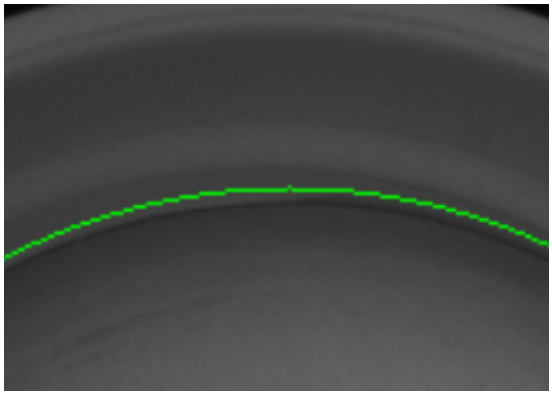


Figure 18: Cap 06, inner OLS results, no blob splitting.

In the context of a linear regression, how can one know which points are influencing the fitting process the most? This points are going to be the pixel outliers, since most of the pixels will produce a similar circle, while these outliers will try to drag the circle too much towards them. Computing the distance between all the points and the circle resulting from the regression process, in order to eliminate the points that are too far away from the circle, is not a good idea in general, since the process minimizes the squared distances and thus a single, very distant point is able to greatly shift the circle up to the point that it doesn't make any sense.
A solution to this is the concept of Cook's distance. Let $s^2$ be the squared error of the model, $k$ the number of regressors, $\mathbf{w_i}$ the weight vector computed without point $i$, the Cook's distance of point $i$ is defined as:

$$D_i = \frac{\|\mathbf{Xw} - \mathbf{Xw_i}\|^2}{ks^2}$$

This distance can be interpreted as a sum of all the squared differences between the output of the model and the output of a model that was fitted without point $i$. Thus, it is a good indicator of how much a single point influences the fitting of the model: points with higher Cook's distance will be more likely to be outliers than points with lower values.
Computing such distances, though, is very heavy: it requires refitting the model as many times as the number of points. Fortunately, it is possible to completely avoid this if we can make use of the pseudo-inverse matrix. Let $\mathbf{X}^\dagger$ be the pseudo-inverse, let $\mathbf{H} = \mathbf{X}\mathbf{X}^\dagger$ be the projection matrix, let $h_{ii}$ be the i-th diagonal element of matrix $\mathbf{H}$, let $e_i = y_i - \mathbf{x_i}^T\mathbf{w}$ be the residual of the i-th point. One can prove that the Cook's distance of the i-th point can be expressed as:

$$D_i = \frac{e_i^2}{ks^2}\frac{h_{ii}}{(1-h_{ii})^2}$$

This formula allows the computation of the Cook's distance by means of a very small set of simple operations, making it very practical to use.

At this point, the idea is to compute the Cook's distance for every edge pixel that survived the blob outliers elimination (figure 19), thresholding the value: a good threshold for this project was found to be 0.0017. After thresholding, it is necessary to fit the model again. Thus, the parameter `circle_generation` that

can already be set to `mean` and `least_squares`, can also be set to `least_squares_cook`, in order to execute this whole process for the generation of the final circle (figure 20).
The images shown in this section are referred to the OLS method with no blob splitting in order to show clearer results, but the options can be combined together (and are actually thought to be used together, since they complement each other).

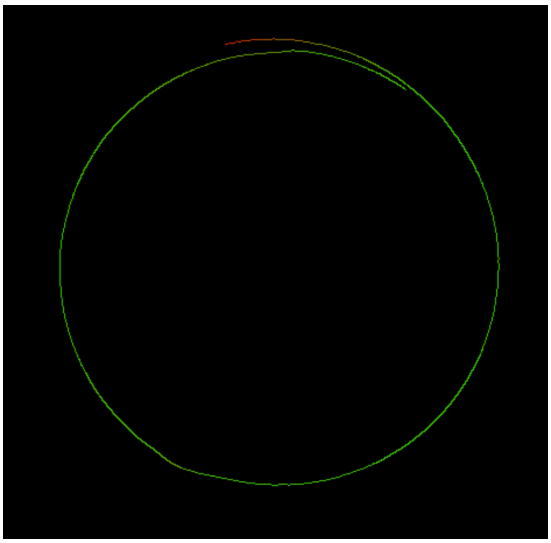


Figure 19: Cap 06, inner edges found with Canny. Red means high Cook's distance.

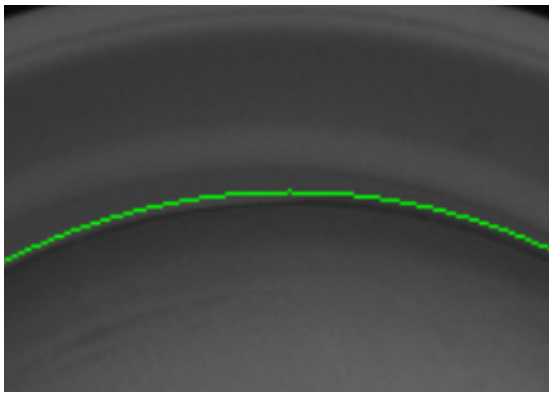


Figure 20: Cap 06, OLS excluding points with high Cook's distance.

Practically speaking, after the blob outliers elimination, the program will first merge the blobs together, it will then fit the model on all of the points, and finally it will compute the Cook's distances. There are two practical implementations of this computation: the first one uses the `statsmodels` library, and is `ols_circle_cook()`; the second one is fully implemented by us and is `fast_ols_circle_cook()`. They basically produce the same results, up to a certain precision, but the second one is much faster, so it is the one we use: for this reason it's not possible to execute the first one by means of a parameter, one needs to directly change the code in `find_circle_ols()` to call that function instead of the other. The fact that our implementation is this faster than the other is probably due to `statsmodels` computing a lot of different things that we don't use, while our function only computes what is needed for the Cook's distance.

Lastly, since the model requires to be fitted again after eliminating some points, and since the computation of Cook's distances needs the pseudo-inverse matrix (the original algorithm as shown in section 2.4, which is slower than R. Bullock's), the final regression is carried out with the Bullock's algorithm.

The best configuration for this method is `outliers_elimination_type = "votes"`, `circle_generation = "least_squares"` and blob splitting set to `true` with minimum dimension 200. Reducing this dimension can produce very imprecise circles due to too few points to fit the model on, eventually harming the voting process, while increasing it means making the process more similar to the one without splitting. As will be clear from the analysis of the last section, the pixel outliers elimination process doesn't provide additional precision in most of the images, while being quite slower.

### 4.3.5 Hough Transform vs Regression Method

Theoretically speaking, the Hough Transform works as follows:

1. Take all the edge points;
2. For each of them, compute all the possible circles that pass though it;
3. Discretize the parameter space, set up a voting process and find a mode value;
4. Possibly compute an average of the best circles around the mode for better precision.

If we consider now the whole method that has been presented in the previous section, without pixel outliers elimination, this is its short summary:

1. Take all the edge points and divide them in groups (blobs);
2. For each group, compute the one and only circle that best fits them (in a least squares sense);
3. Discretize the parameter space, set up a voting process and find a mode value;

4. Extract the groups that voted for the winning value, possibly compute an average (actually another OLS fit) for better precision.

It is evident now that the regression method that we produced is quite similar to the Hough Transform: in fact, it is basically a blob-wise version of the normal pixel-wise Hough Transform and produces very similar results in a fraction of the time. The main difference, besides this, is that the mode value of the HT parameter space is the resulting circle itself, whereas the mode value of the regression method contains multiple blobs that can be fitted again together. This also implies that the resolution of the parameter space can be much lower for the regression method, speeding it up. Moreover, thanks to the blob-wise computation, a blob only votes for one circle, eliminating *de facto* the burden of the accumulator array and allowing the usage of a dictionary instead.
Considering cons, instead, it is also clear that the main limitation of the regression method is in the blob division itself: a fit on a blob is quite imprecise because its points are connected together and form a small part of the circle, and thus single blobs can easily vote for very different circles even if they actually belong to the same circle, and this is the exact reason for the resolution factor to be so high. This means that the precision of the voting process is reduced, and it may fail to separate 2 close circles. If one tries to let the algorithm find these 2 circles by decreasing the resolution factor (i.e. increasing the resolution), it may actually end up finding a single and very bad circle, because the different blobs all vote for slightly different circles and most of them get excluded. These facts didn't cause problems in this project, although they can be seen by displaying the remaining blobs after the voting process, but if one wants to use this algorithm in general they are to take into account. A possible way to deal with this problems can be making the votes Gaussians instead of points, thus finding "hot areas" in the parameter space, whose resolution can be increased.

# 5 Outlining the defects

A defect consists, at least in the provided examples, in a straight line from border to border of the inner circle, indicating that the liner is present but not complete. The hint for defect detection and outlining consists in computing the magnitude of the gradient for each pixel and in fact, because of the lighting of the image and because of the different average brightness of the liner and the missing liner, the edge of defective liner is dark, as opposed to the light liner and the even lighter missing liner. The idea is then to compute the magnitude of the gradient and threshold it, but the already widely used Canny's edge detection algorithm already has all the necessary steps, providing also resistance to light shifts thanks to the hysteresis thresholding. Since it does all of this by computing the magnitude of the gradient, it is consistent with the project hint.

Starting from the stretched image, we proceed to apply a 7x7, $\sigma = 2$ Gaussian filter for the same reasons of the previous section, and then we execute the `Canny()` function. Since the defect is located inside the inner circle, the output of the function is masked by means of a circular mask with the same center of the inner circle and a radius that is 95% of the inner circle, in order to cut out the edge points that produced the circle (figure 21 and 22).

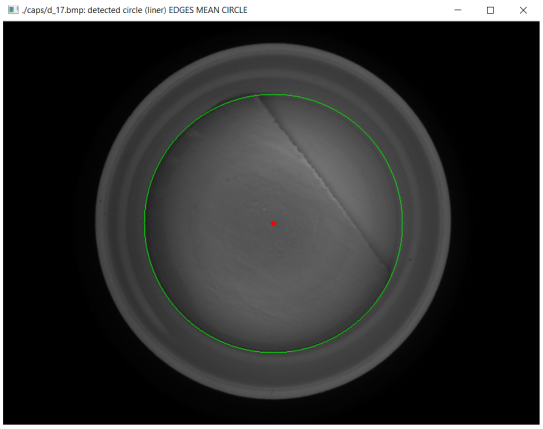



Figure 21: Defective cap 17 with its inner circle.

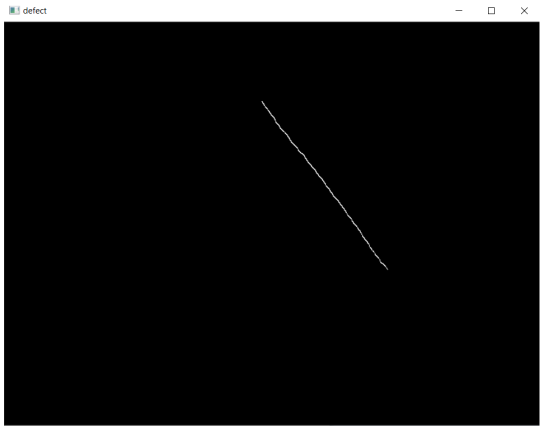



Figure 22: Internal edge points found with `Canny()`.

At this point we end up with a set of points and we need a way to tell if this set of points makes up a defect or not. In terms of test images, the presence itself

of one single point or more already indicated a defect with 100% recall and great precision. Precision reaches 100% by simply setting a (small) threshold for the minimum number of edge pixels found, but this method doesn't seem to be robust. There are multiple ways to proceed, for example: computing a Hough Transform for lines and thresholding the results to be sure that enough points voted for a certain line; computing the correlation coefficient of the points to check for linearity; etc. Another way consists in checking that the found edge touches the inner circle in 2 distinct points, producing a separation of the inner pixels that can lead to the computation of 2 different average lightnesses, which will differ by a certain amount.
Since we don't have enough examples, it's not clear whether we should choose robustness and rigidity (with lots of requirements and controls to be satisfied) or simplicity (since, given the environment, all the defects will be very clearly recognizable), so we actually chose a compromise. We implemented the last algorithm described, but without the computation of the averages. Thus, having the masked edges, we proceed by drawing a circle with the same parameters of the mask, and then we compute the intersection between this circle and the edge points. Because of the discrete plane and the connectivity limitations of pixels, it's possible that the edge points (virtually a line) cross the circle diagonally without having points in common with it. In order to solve this problem, the circle is actually drawn with a thickness value of 2.
The problem that arises from a thick circle is that having 2 intersection points is no more a valid check for the line to be crossing the theoretical circle in 2 points, since they could be both in the same side of the edge line. Thus, we compute all the distances between pairs of intersection points, that are very limited in number, and check that at least one of them is higher than $\frac{1}{10}$ of the radius of the inner circle. If this is verified, it is considered a valid defect, and is enclosed in a rectangle by means of OpenCV's `minAreaRect()` function. This rectangle is then manually enlarged in its lower dimension in order to make it more visible in the images (figure 23).

The aforementioned method yields perfect precision and recall scores on test images, but of course has its limitations. On the one hand it could be more robust, for example by computing the averages or by checking the linearity of the edge points, which could be easily done also by checking the number of such points

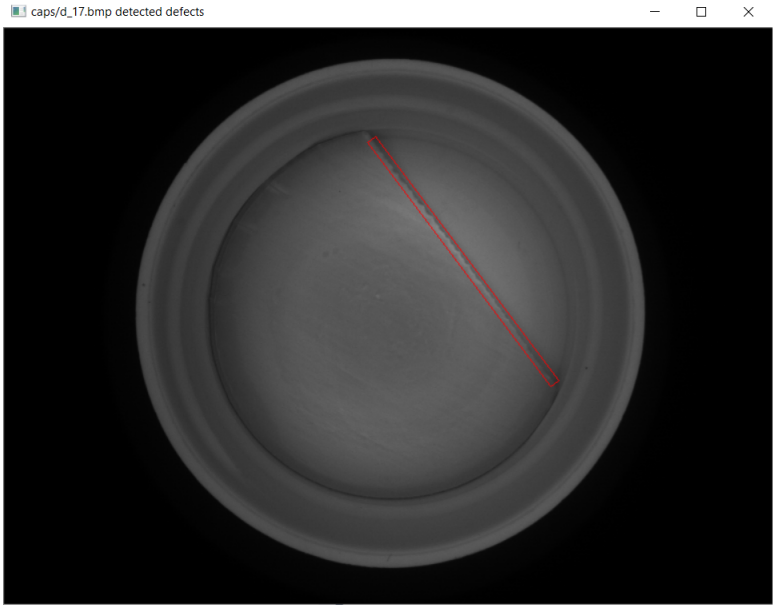



Figure 23: Cap 17, enclosing rectangle of the defect.

against the bigger dimension of the enclosing rectangle. On the other hand, it could also be simpler if there is no need of such precision.
Moreover, this method is making 2 precise assumptions: all the edge points of a defect are connected together; they are long enough to touch the circle in 2 points. If a single pixel is missing, splitting the defect in 2, or if an edge is not clear and isn't fully picked up by Canny's algorithm, it won't be recognized as such.

Conscious of the comprises and limitations of this method, we decided to implement it in the described way with function `defects_enclosing_rectangles()` in `program.py`, which requires as inputs the image and the information about the inner circle, in order to mask the edges. Provided more test images, it is possible to make it more robust or to make it simpler, if necessary.

Code Snippet 6: `program.defects_enclosing_rectangles()`

```
1  def defects_enclosing_rectangles(img, liner_xc, liner_yc, liner_r):
2
3      gaussian = cv2.GaussianBlur(img, (7,7), 2, 2)
4
```

```
    # Outline edges and delete what is outside the liner, then compute connected blobs.
    edges = cv2.Canny(gaussian, 20, 110, apertureSize=3, L2gradient=True)
    mask = utils.circular_mask(img.shape[0], img.shape[1], (liner_xc, liner_yc), 0.95*liner_r)
    edges[~mask] = 0

    has_defects = False
    blobs = utils.get_blobs(edges)

    # Draw a circle, slightly smaller than the liner.
    liner = np.zeros((img.shape[0],img.shape[1]), dtype=np.uint8)
    cv2.circle(liner, (np.round(liner_yc).astype("int"), np.round(liner_xc).astype("int")), np.round(0.95*liner_r).astype("int"), (255, 255, 255), 2)
    nonzero = np.nonzero(liner)
    liner = list(zip(nonzero[0],nonzero[1]))

    rectangles = []

    for blob in blobs:
        # Intersections with the circle
        common = list(set(liner).intersection(list(zip(blob[0],blob[1]))))

        # Compute the maximum distance between intersection points.
        max_distance = 0
        for pixel in common:
            for pixel2 in common:
                distance = math.sqrt((pixel[0]-pixel2[0])**2 + (pixel[1]-pixel2[1])**2)
                if distance > max_distance:
                    max_distance = distance
```

```
32
33          if len(common) >= 2 and max_distance > liner_r/10:
34              has_defects = True
35              rect = cv2.minAreaRect(np.array(list(zip(blob[0],
                 ↪  blob[1]))))
36              rect_dim = rect[1]
37              # Increase the smaller dimension of the rect, to make it
                 ↪  more visible.
38              if rect_dim[0] < rect_dim[1]:
39                  rect_dim = (rect_dim[0]*2, rect_dim[1]*1)
40              else:
41                  rect_dim = (rect_dim[0]*1, rect_dim[1]*2)
42              rect = (rect[0], rect_dim, rect[2])
43              box = cv2.boxPoints(rect)
44              box = np.int0(box)
45              rectangles.append(box)
46
47      return has_defects, rectangles
```

# 6 Performances and results

We present here a brief description of the results, both in terms of precision and performance.
For the outer circle, basically all the possible methods will be shown. For the inner circle, instead, they are definitely too many: we will show only the most important ones.
Performances were measured by means of Python's `time.perf_counter()` function. Performance optimization and analysis was possible thanks to the line profiling tool by Robert Kern[6].
In the next tables, the performance indexes (PI) are computed with respect to a reference method, having an index equal to 1x. A PI equal to 2x means that the method is twice as fast as the baseline.

## 6.1 Linear regression

Here is a comparison of the two algorithms for least squares linear regression: the normal pseudo-inverse algorithm and Bullock's algorithm. 10 different sets of random points were generated for each set size, and both the algorithms were executed on these sets for 1000 times. The execution times shown are the average times for the methods to be executed on all the 10 sets and are expressed in seconds, and thus the time for the algorithms to be executed on a single set is just $\frac{1}{10}$ of what is shown here.

The following results were obtained with an Intel i5-4258U processor at 2.90 GHz, on a Windows 10 platform with Python 3.6.6 and opencv-python 4.0.0.21.

Table 1: Execution times and PI for least squares linear regression algorithms (Intel i5-4258U, 1000 iterations).

| Method\Number of points | 100 | 1k | 10k | 100k |
|---|---|---|---|---|
| OLS pseudo-inverse | 0.001940 s | 0.002278 s | 0.005340 s | 0.075655 s |
| Bullock's OLS [1] | 0.000455 s | 0.000629 s | 0.001859 s | 0.038550 s |
| PI of Bullock's OLS | 4.3x | 3.6x | 2.9x | 2.0x |

[6]https://github.com/rkern/line_profiler

The following results were obtained with an Intel i7-8750H processor at 4.10 GHz, on a Windows 10 platform with Pytnon 3.6.6 and opencv-python 4.1.0.25.

Table 2: Execution times and PI for least squares linear regression algorithms (Intel i7-8750H, 1000 iterations).

| Method\Number of points | 100 (s) | 1k (s) | 10k (s) | 100k (s) |
|---|---|---|---|---|
| OLS pseudo-inverse | 0.000820 s | 0.000997 s | 0.003222 s | 0.036360 s |
| Bullock's OLS [1] | 0.000416 s | 0.000540 s | 0.001353 s | 0.019751 s |
| PI of Bullock's OLS | 2.0x | 1.8x | 2.4x | 1.8x |

The edge of Bullock's algorithm is very clear in these tables, being solidly twice as fast as the pseudo-inversion method. The advantage of this algorithm is less evident with the faster processor, and tends to decrease with the size of the problem, but it is still very evident even for 100k points, which is well above the usual number of points we deal with (200 - 3k). From the implementation viewpoint, it is also worth noting that the pseudo-inverse method consists in the direct usage of Numpy's matrix inversion function, while the implementation of Bullock's algorithm can be slightly slowed down due to multiple calls to different Numpy functions.

For this reason, the algorithm used for OLS computations throughout the project is Bullock's, with the exception of those methods that require the computation of Cook's distance, which requires the pseudo-inverse matrix.

## 6.2 Cook's distance computation

Here we compare the `statsmodels` implementation (`ols_circle_cook()`) against our own method (`fast_ols_circle_cook()`). Since the former was very slow, only 100 iterations were carried out, in the same way as before (10 random sets, overall time).

The advantage of our implementation is very clear, independently from the CPU and the set size (around 30 times faster) and thus it was deployed in the project. This big difference, as already stated, can be due to `statsmodels` computing a lot of different things automatically, besides the Cook's distance.

Table 3: Execution times and PI for OLS with Cook's distance computation (Intel i5-4258U, 100 iterations).

| Method\Number of points | 10 | 100 | 1k |
|---|---|---|---|
| Statsmodels | 0.620671 s | 0.413210 s | 5.464701 s |
| Pseudo-inverse Cook | 0.004636 s | 0.013915 s | 0.145801 s |
| PI of pseudo-inverse Cook | 134x | 29.7x | 37.5x |

Table 4: Execution times and PI for OLS with Cook's distance computation (Intel i7-8750H, 100 iterations).

| Method\Number of points | 10 | 100 | 1k |
|---|---|---|---|
| Statsmodels | 0.055680 s | 0.278134 s | 3.501862 s |
| Pseudo-inverse Cook | 0.002370 s | 0.008286 s | 0.095276 s |
| PI of pseudo-inverse Cook | 23.4x | 33.6x | 36.8x |

The first PI of the i5 processor is definitely too high, and could be due to loading times for the first execution of the `statsmodels` library, but we were not able to have it correct.

## 6.3 Outer circle

In the Excel file /tests/Results/precision.xlsx in the repository, the main methods were tested for precision in a scale from 0 to 10 for every image. A 10/10 mark means that the circle is perfect, while a mark above 8/10 means that it is still totally acceptable. An example of mark 8 is figure 18, which is perfect except for the shown region, while figure 20 is a 9 for the same reason. Since we have 16 test images, the total score for each method is given by the sum of its scores. In the following tables we only report the sum.
The execution times are referred to the computation being carried out on all the 16 test images after that each of them has been loaded, pre-filtered and prepared for computation, so that the times take into account the specific method only.

[7]Hough Transform, averaging the 3 best circles.

Table 5: Execution times, PI and precision scores for outer circle detection methods (100 iterations).

| | i5-4258U | | i7-8750H | | |
|---|---|---|---|---|---|
| Method | Time | PI | Time | PI | Precision score |
| HT-AVG3[7] | 4.040514 s | 1x | 1.783946 s | 1x | 159/160 |
| HT-AVG3-BD[8] | 4.113036 s | 0.98x | 1.856192 s | 0.96x | 160/160 |
| **OLS-MEAN[9]** | **0.084907 s** | **47.6x** | **0.046120 s** | **38.7x** | **160/160** |
| **OLS-OLS[10]** | **0.085955 s** | **47x** | **0.047405 s** | **37.6x** | **160/160** |

As a reference, the pure Hough Transform taking the best circle only, and without any averaging, got a score of 144/160.

From this table it is clear that the regression methods perform perfectly, while having 30-50 times faster performances, and thus are the preferred methods (with all the already specified limitations).

## 6.4 Inner circle

The same table as before is presented for the inner circle too, with the times not including pre-processing operations and outer circle computation (needed to compute the inner circle), but including all the method-specific operations such as Gaussian filters, linear stretching and edge detection.

OLS-MEAN-MEAN is in the table for reference only, and is missing precision

[8]HT averaging the 3 best circles and computing the radius by averaging the distances from the perimeter points.

[9]Least squares linear regression on each blob, with the final result being a weighted mean of the regressions.

[10]Least squares linear regression on all the points.

[11]Hough Transform, averaging the 2 best circles.

[12]HT with L2 Canny's edge detection, averaging the 2 best circles.

[13]Least squares linear regression on each blob, outliers elimination with the "mean" method and the final result being a weighted mean of the regressions.

[14]Performing OLS on each blob, outliers elimination with the voting process and final result being a regression of all the remaining points.

[15]Splitting the blobs with minimum size equal to 200, performing OLS on all the blobs, outliers elimination with the voting process and final result being a regression of all the remaining points.

Table 6: Execution times, PI and precision scores for inner circle detection methods (100 iterations).

| | i5-4258U | | i7-8750H | | |
|---|---|---|---|---|---|
| Method | Time | PI | Time | PI | Precision score |
| HT-AVG2-NORMAL[11] | 1.964523 s | 1x | 0.878535 s | 1x | 129/150 |
| HT-AVG2-PRECISE[12] | 0.977329 s | 2x | 0.484445 s | 1.8x | 147/150 |
| OLS-MEAN-MEAN[13] | 0.391200 s | 5x | 0.201652 s | 4.4x | - (bad score) |
| OLS-VOTE-OLS[14] | - | - | - | - | 146/150 |
| **OLS-S200-VOTE-OLS**[15] | **0.398432 s** | **4.9x** | **0.218151 s** | **4x** | **148/150** |
| OLS-S200-VOTE-COOK[16] | 0.689762 s | 2.8x | 0.406241 s | 2.2x | 144/150 |

results because they were too bad. OLS-VOTE-OLS is missing execution times because they are basically identical to OLS-MEAN-MEAN and OLS-S200-VOTE-OLS.

Due to the multiple steps required by the regression methods compared to the simple regression carried out for the outer circle, the performance difference is not as high as before. Nevertheless, it is very clear that regression methods are much faster while having similar precision, and actually better in some cases. Moreover, the speed of such methods can be increased by a good margin, since they are currently implemented in Python + Numpy while the HT is implemented in C++. All the methods scoring 144/150 or more are considered good, with no images having lower scores than 8/10. The best method found is then OLS-S200-VOTE-OLS, which is the procedure described in section 4.3.5 and can be seen as a "blob" version of the Hough Transform.
The last line of the table also adds pixel outliers elimination by thresholding the Cook's distance, producing a quite slower performance due to the already analysed reasons. In this project, adding this procedure actually worsen the precision scores, but it seems to have a theoretical foundation and thus it may end up making things look better in different situations.

[16]Splitting the blobs with minimum size equal to 200, performing OLS on all the blobs, outliers elimination with the voting process, elimination of pixels with a high Cook's distance and final result being a regression of all the remaining points.

It's also worth noting that the HT method with an additional explicit Canny's edge detection is actually faster than the normal HT, even if the edge detection is carried out twice, because it greatly reduces the number of edge points.

# 7 Conclusions

Overall, we were able to achieve perfect results for the outer circle, and very good results for the inner circle too, with also great performances in terms of execution time. The methods we developed showed great results (faster and more precise) and seem to be robust, but we would suggest further testing in order to assess if they can actually be deployed in substitution of the well proven Hough Transform. Additionally, our defect detection method can be considered quite arbitrary, as it is a result of a compromise between precision (implying a strict algorithm) and recall requirements (implying a loose algorithm) and our small number of testing images.

Moreover, in the development of our methods, we tried to be as much invariant as possible. Translation should not be a problem by any means, just like rotation. Scaling is more problematic, but we tried to partially achieve invariance by making no assumptions for the outer circle detection and, for the inner circle, basing our assumptions on the previously determined outer circle only. Basically, the only non scale-invariant parameters in the model are the dimensions of the filters and the minimum dimension of the blobs for blob splitting. Overall, we would say that the whole program is scale invariant *up to a certain scale*.

Speaking of quantitative results, the Bullock's algorithm has been proven to be quicker than a pseudo-inverse matrix computation, and we were also able to produce a quicker Cook's distance computation with respect to `statsmodels`. As expected, regression is much faster than the Hough Transform and it can be easily deployed for the outer circle (with some assumptions), while the "blob-wise" version of the Hough Transform provides very accurate and fast results for the inner circle.

# References


[1] R. Bullock. *Least-squares circle fit*. `https://dtcenter.org/met/users/docs/write_ups/circle_fit.pdf`. 2006.

[2] Wan-Yu Chang, Chung-Cheng Chiu, and Jia-Horng Yang. "Block-Based Connected-Component Labeling Algorithm Using Binary Decision Trees". In: *Sensors (Basel, Switzerland)* 15.9 (Sept. 2015), pp. 23763–23787.

[3] Raul Santiago-Montero, Ernesto Bribiesca, and R Santiago. "State of the art of compactness and circularity measures". In: *International Mathematical Forum* 4 (Jan. 2009), pp. 1305–1335.

[4] H. K. Yuen et al. "Comparative study of Hough Transform methods for circle finding". In: *Image Vision Comput.* 8 (1990), pp. 71–77.