Department of Engineering and Physical Science

Robotic System Science

Assignment 3 – Computer Vision

JANNIN Sylvain

H00387879

Summary

Table of figures

1. Part 1 – Object detection

The goal of this assignment is to discover and learn how to use tools about image processing by finding a predefined object in a picture. We chose a picture of coins with different sizes and colours. We used OpenCV on Python because it is a very useful and easy tool for image processing with a lot of documentation. To load the image on Python we used *imread* function.

Figure 1: picture we choose.

The second step is to convert the image on grey and on Hue, Saturation, Value (HSV). For the grey image we used a parameter in *imread* function to have it. For the HSV picture we use *cvtColor* with the parameter *COLOR_BGR2HSV* and then we split it.

Figure 2: grey picture.

Figure 3: picture divided in Hue, Saturation and Value.

Since the second picture above (the saturation one) is black if it is not a coin, otherwise it is grey if it is a coin. So, we with the picture we already have a good shape, and it is going to be easy to keep only the coins. Therefore, we decided to use the Saturation picture for the following steps. Now we need to print the histogram of this picture and apply a threshold on it in order to binarize the picture.

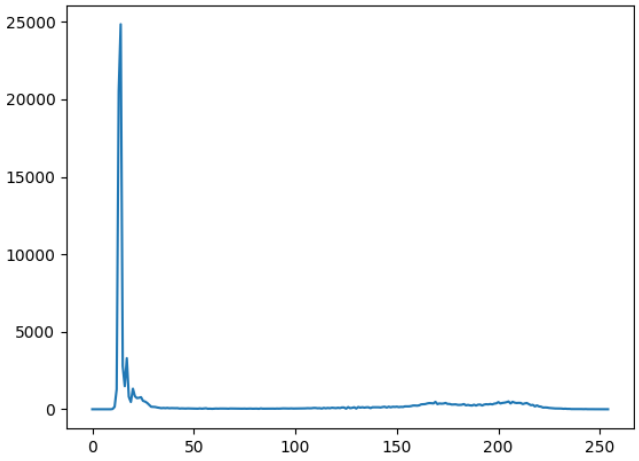


Figure 4: histogram of the chosen image.

To binarize the image we applied a threshold from 75 to 255. It created a picture where every pixel under 75 are set to 0 (white), and the rest to 255 (black).

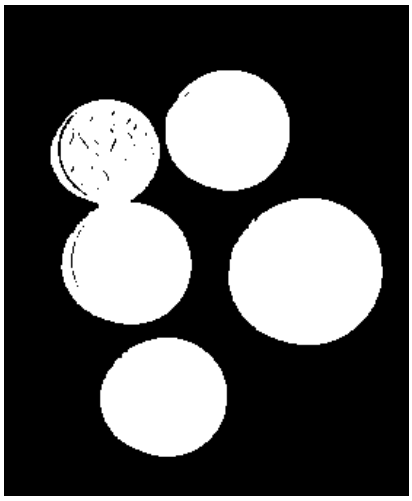


Figure 5: current picture with the threshold applied.

On this image we can observe noises (the black pixels on the in the coins), especially in the top left coin. To delete these black pixels, we did a dilation thanks to *dilate* function. The goal of dilation is, for each white pixel, dilate around this pixel, in other word if the pixel is white all around this pixel, we add white pixel.

Nevertheless, only doing a dilation will false the image since the edges of our coins are going to dilate. To correct this, we did an erosion with *erode* function. The erosion process is the opposite of the dilation. Thanks to that, we can keep the same shape for our coin edges.

Only one iteration is enough to take off the noises. We have the same iteration number for both dilation and erosion and same kernel (a 3x3 matrix).

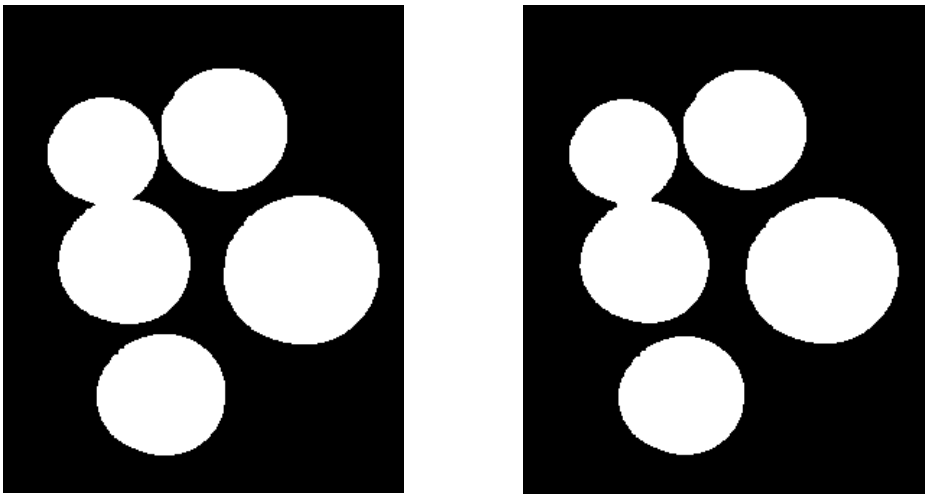


Figure 6: dilation then erosion.

Before detecting the edge. We did a the smooth the picture with *GaussianBlur* function with a 5x5 matrix for the kernel. The goal is to take off some noises that could have been remain. Doing a gaussian filter is a commonly recommended step before doing an edge detection.

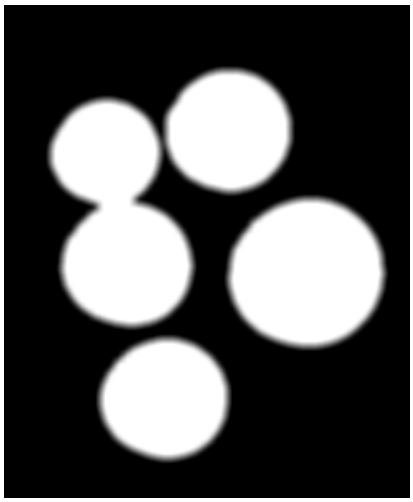


Figure 7: gaussian smoothing.

Once the smoothing was done, we use *Canny* function to do the edge detection.

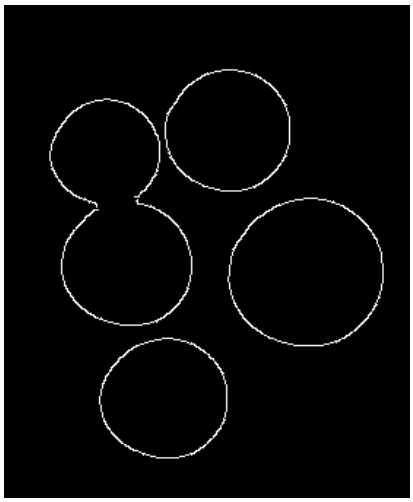


Figure 8: edge detection.

We successfully managed to detect the edges of our coins. We can notice that two coins are more stick on the edge detection than on the original picture.

2. Part 2 – Camera calibration

2.1. Calibration process

To calibrate the camera the first step is to take pictures of a checkboard with different angle and distance. we used the one in the GRID, we took picture of it with our phone.

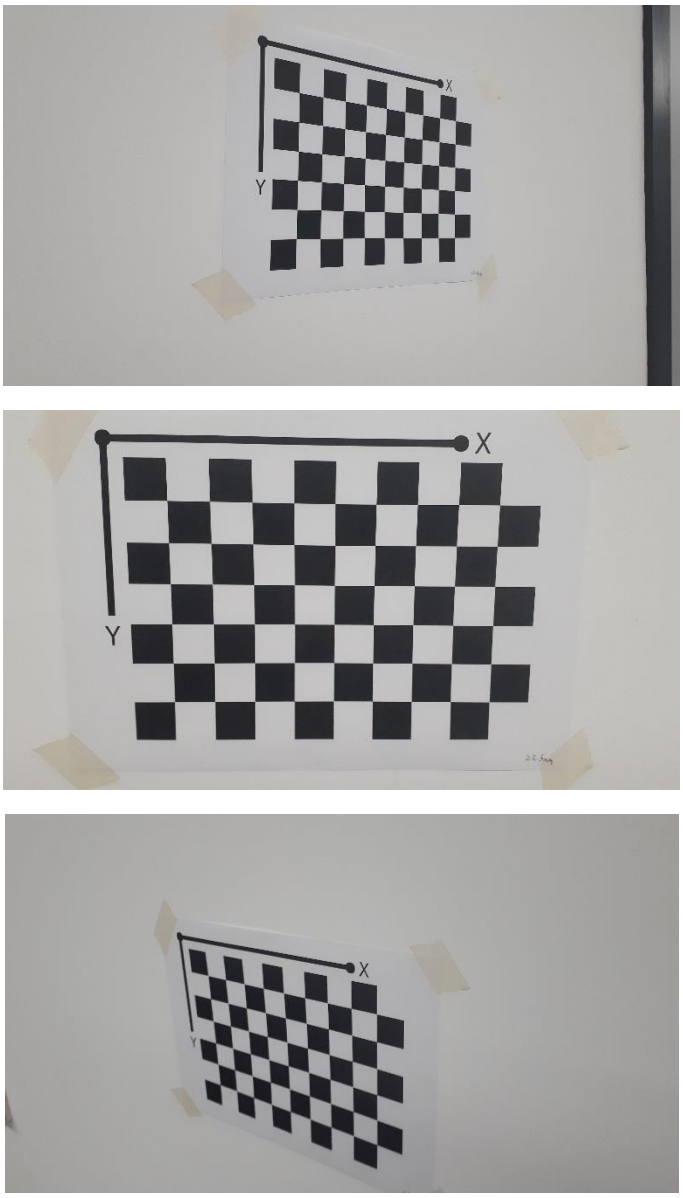


Figure 9: pictures of checkboard on different angles and distance.

Then, to calibrate the camera used *camera calibrator* which is an app on Matlab. We uploaded the pictures we took. The software first asked the size of the checkboard, in our case it was 22.5 mm. It is recommended to upload at least 10 images, in our case we uploaded 26 images.

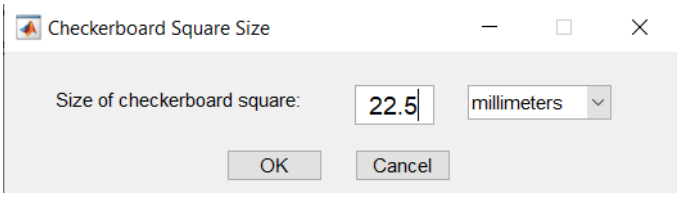


Figure 10: size check on Matlab.

After Matlab processed the images, the app prevents how many and which images are rejected. In our case 2 images has been rejected, these two pictures were too much bend to the checkboard.

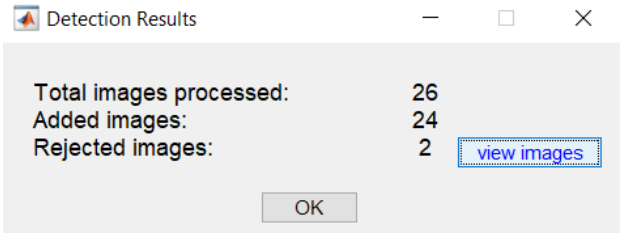


Figure 11: result of the process.

With that, we can calibrate the camera and export the camara parameters. When the calibration is done, we can see several information gave by the app like the *Reprojection Errors* or the angle where the pictures have been taken from for the calibration.

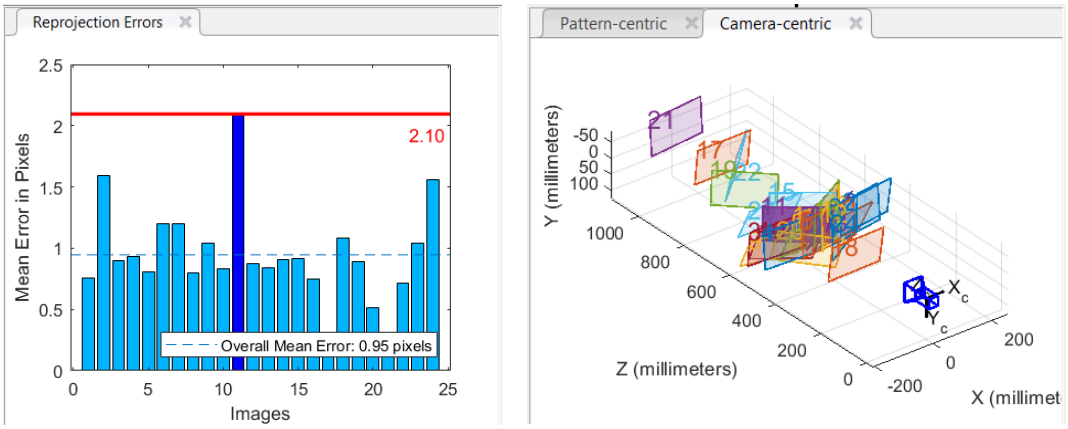
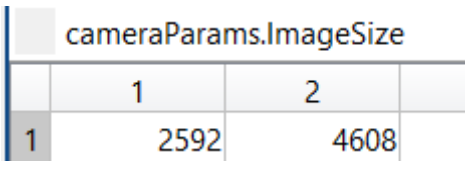


Figure 12: information given after the calibration.

The main goal of the calibration is to export the parameters and to obtain the image size and the camera *intrinsic matrix*.



cameraParams.ImageSize

	1	2
1	2592	4608

Figure 13: size of the image from Matlab.

cameraParams_Homework2.IntrinsicMatrix

	1	2	3
1	3.6911e+03	0	0
2	0	3.6860e+03	0
3	2.2881e+03	1.2947e+03	1

Figure 14 : camera intrinsic matrix from Matlab.

2.2. Camera field of view

Once we have the intrinsic matrix, we can calculate the field of view of our camera. At the left we have the position of a pixel on a picture and at the right the position in real space. Since M_{ex} is a 3x4 matrix, if we multiply this matrix with the vector, we have a 3x1 vector where each point of the result is the position of the point in real life.

$$\begin{bmatrix} x_h \\ y_h \\ w \end{bmatrix} = M_{in} M_{ex} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x_h \\ y_h \\ w \end{bmatrix} = M_{in} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x_h \\ y_h \\ w \end{bmatrix} = M_{in} \begin{bmatrix} X_w \\ Y_w \\ Z_w \end{bmatrix}$$

$$M_{in}{}^{-1} \begin{bmatrix} x_h \\ y_h \\ w \end{bmatrix} = \begin{bmatrix} X_w \\ Y_w \\ Z_w \end{bmatrix}$$

With (the matrices have been rounded):

$$M_{in} = \begin{bmatrix} 3691 & 0 & 2288 \\ 0 & 3685 & 1294 \\ 0 & 0 & 1 \end{bmatrix}$$

$$M_{in}{}^{-1} = \begin{bmatrix} 2.709 \times 10^{-4} & 0 & -6.198 \times 10^{-4} \\ 0 & 2.712 \times 10^{-4} & 3.512 \times 10^{-4} \\ 0 & 0 & 1 \end{bmatrix}$$

Once we have the position in real world. We can use them to calculate the vertical, horizontal and diagonal field of view. The points we chose are opposite regarding the centre of the image. P_1 and P_2 are used for the vertical field of view.

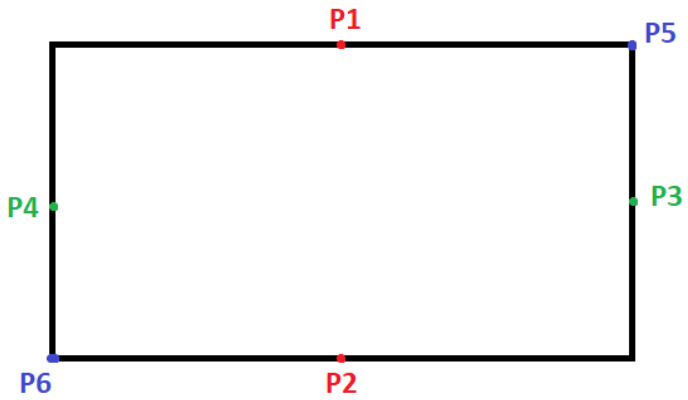


Figure 15: position of the point we used to calculate the fov.

To calculate field of view we need the position of the point and their norm. we call v_n the vector of point *n* and fov the field of view calculated.

$$v_n \,.\, v_{h+1} = \|v_h\| \times \|v_{h+1}\| \times \cos\,(fov)$$

$$fov = arcos\left(\frac{v_n \,.\, v_{h+1}}{\|v_h\| \times \|v_{h+1}\|}\right)$$

These are the general formulas, to calculate this we created a python program. We entered the useful variables: intrinsic matrix and picture size in pixel. With this, we can compute the pixel positions and their 3D positions. The function *to3d* is used to calculate a pixel in 3D position and *calculate_fov* is used to compute the field of view.

For example, for P_1 and P_2, the picture has 4608 pixels on the height axis and 2592 on the width axis.

$$P_1 = \left(\frac{\text{height}}{2}, 0{,}1\right); P_2 = \left(\frac{\text{height}}{2}, \text{width}\ ,1\right)$$

$$P_1 = (1533, 0, 1); P_2 = (1533, 2592\ ,1)$$

$$v_1 = {M_{in}}^{-1}\begin{bmatrix}1533\\2592\\1\end{bmatrix};\ v_2 = {M_{in}}^{-1}\begin{bmatrix}1533\\2592\\1\end{bmatrix}$$

$$v_1 = {M_{in}}^{-1}\begin{bmatrix}1533\\2592\\1\end{bmatrix};\ v_2 = {M_{in}}^{-1}\begin{bmatrix}1533\\2592\\1\end{bmatrix}$$

$$v_1 = \begin{bmatrix}0.0043\\-0.3512\\1\end{bmatrix};\ v_2 = \begin{bmatrix}0.0043\\0.3512\\1\end{bmatrix}$$

For the field of view we simply applied the equation above with the dot product, results have been rounded thanks to *np.round*.

```
Field of view in degree :
Vertically   : 39.0
Horizontally : 64.0
Diagonally   : 71.0
```

Figure 16: vertical, horizontal and diagonal field of view.

2.3. Calculate distance

In the following figure, there is a schematic of the current question. The goal is to calculate with our camera, how far we should be at least to take a picture of 10m hight tree.

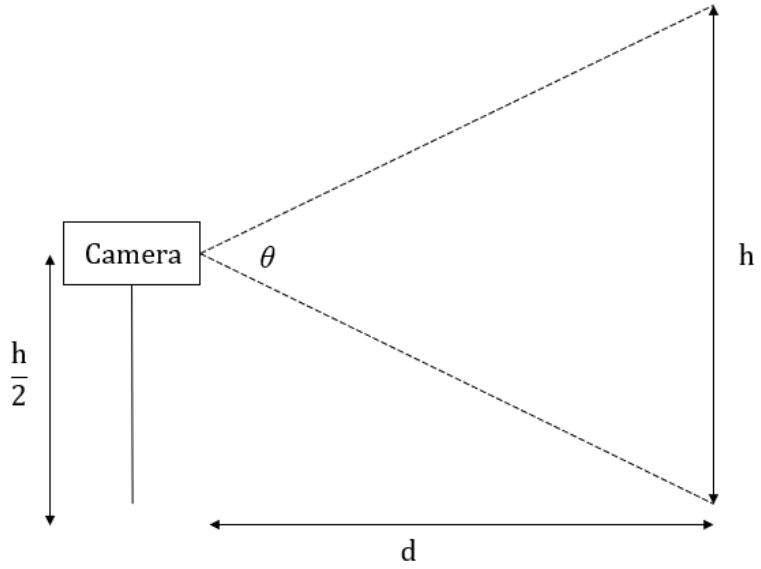


Figure 17: question 2 schematic.

In this case, it is just a simple geometric problem. θ is our vertical field of view.

$$d = \frac{\frac{h}{2}}{\tan\left(\frac{\theta}{2}\right)}$$

$$d = \frac{\frac{10}{2}}{\tan\left(\frac{39}{2}\right)}$$

$$d \approx 14{,}22 \text{ m}$$

With our camera, the user must be far from the tree of at least 14.22 meters.

3. Part 3 – calculating distance between a camera and an object

For the last part, the goal is to detect and predefined object in a picture and to find the distance between this object and the camera. The chosen picture is the one below. For this, we used the same cameras as the previous part and the object is the clementine. We assumed to simplify calculous that the clementine is a sphere of 4.5cm diameter.

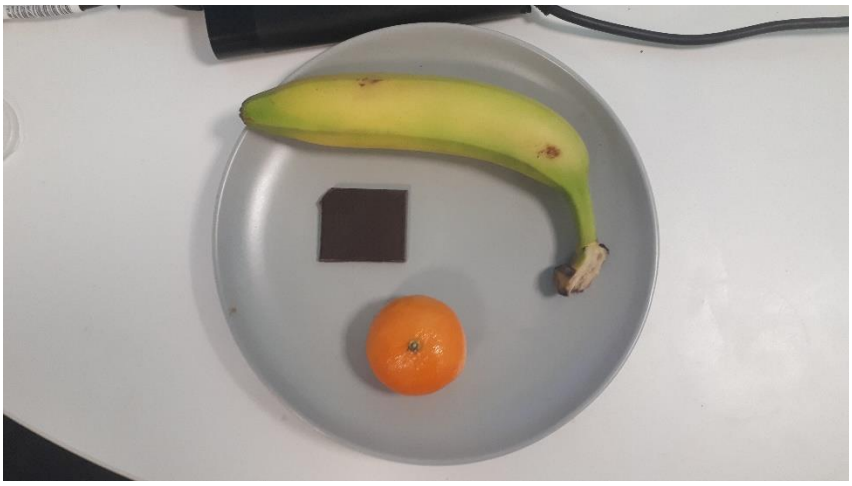
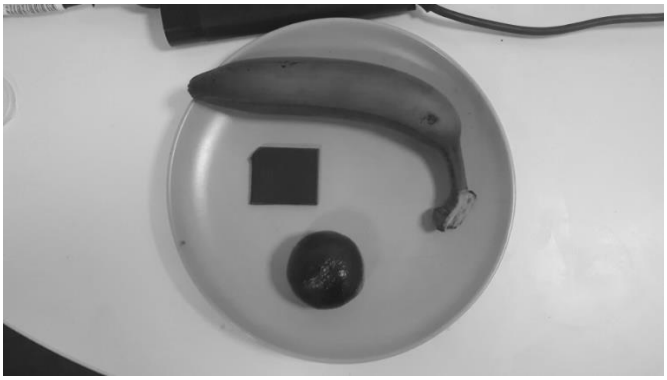


Figure 18: chosen picture for part 3.

3.1. Object detection

To detect the object, we did the same thing as the first part: opening the image and see which image is the easiest to work with. This time, the image is bigger than the coins picture in the first part, so we need to resize the picture by using *resize*, we only keep 20% of the picture.



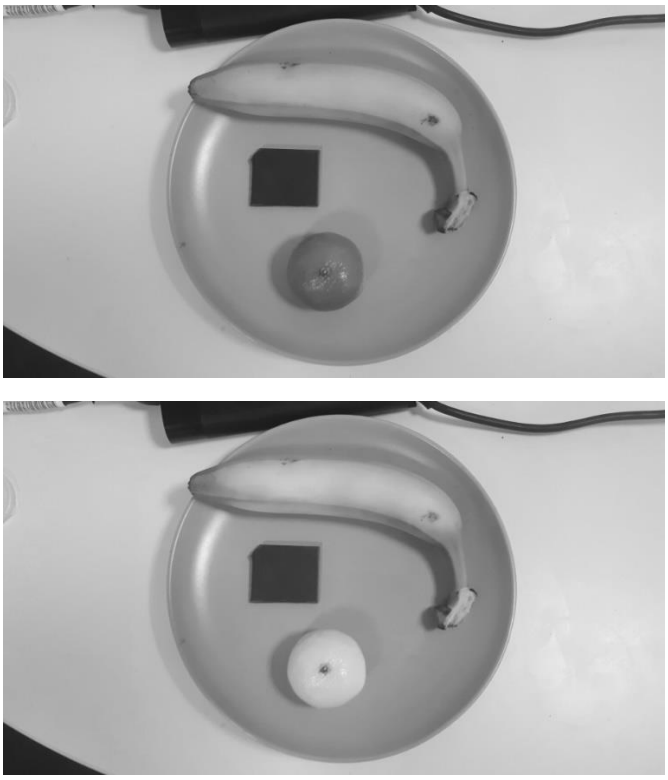
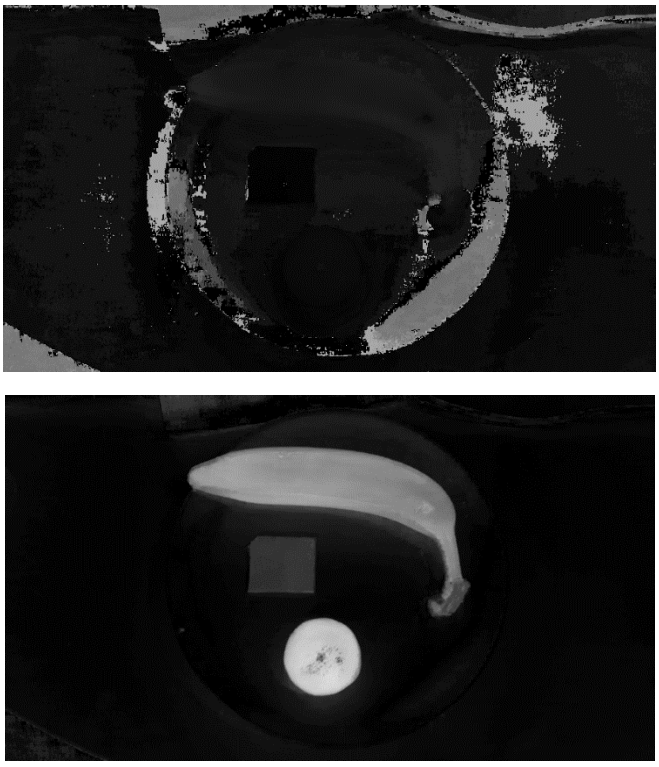


Figure 19: picture divided in blue, red, green.



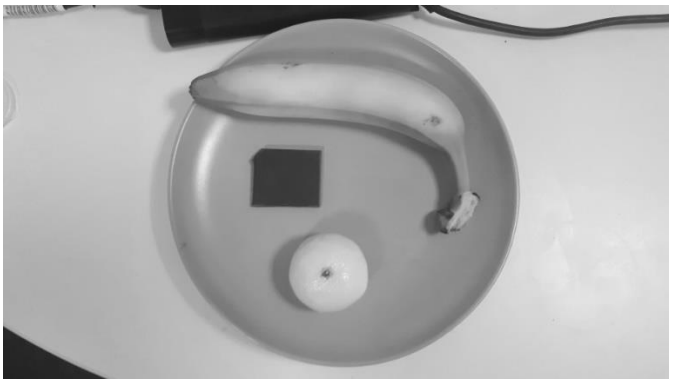


Figure 20 : picture divided in Hue, Saturation and Value.

As for the first part, the easiest one to work on is the saturation one. Then we plotted the histogram to find a good threshold.

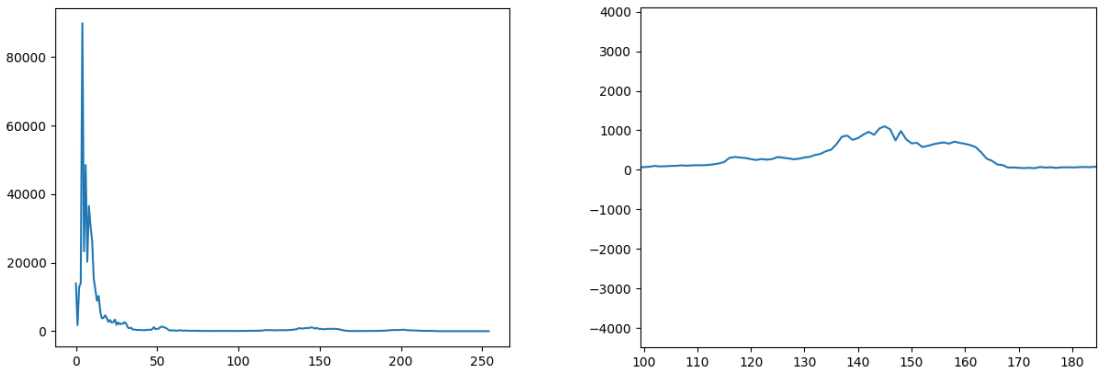


Figure 21: Histogram of the saturation picture.

A good threshold would have been for 0 to 130 but with the threshold we still have the banana as we can see on the figure below. Even if the shape of the clementine is good, we need to take a bigger threshold to take off the banana.

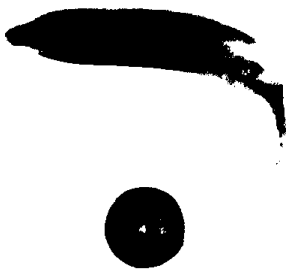


Figure 22: binarized image of a threshold from 0 to 130.

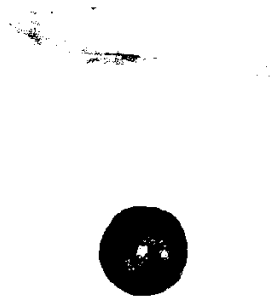


Figure 23 : chosen binarized image with a threshold from 0 to 164

Once with have a binary image, we must take off the noises and then apply a gaussian filter on it. To take off the noise we do a dilation for 2 iterations and an erosion for 4 iterations.

Figure 24: dilation and erosion.

Figure 25: image with a gaussian smoothing.

Now we can use *Canny* to do the edge detection on our image which is showed in the next figure. Even if there are noises in the centre or the clementine, it is not annoying to detect the object.

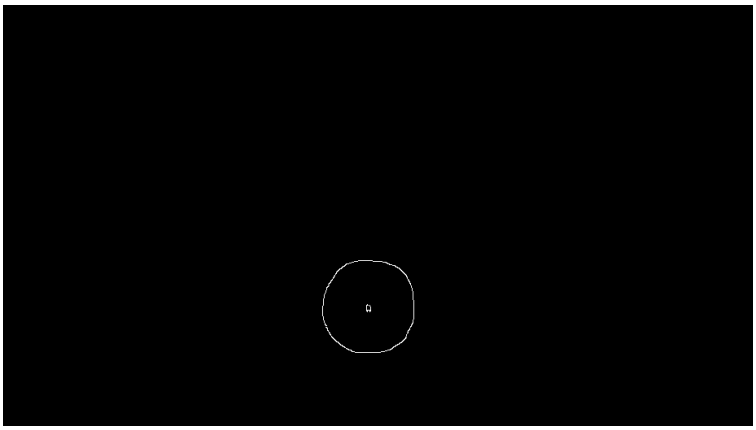


Figure 26: edge detection of the clementine.

3.2. Calculate the distance

Our picture was resized to 20%, height has now 518 pixels and width had 921 pixels. To calculate ΔP, we created the function *get_pixels* which find the leftmost and rightmost pixel on the width axis, then we subtract them to have ΔP.

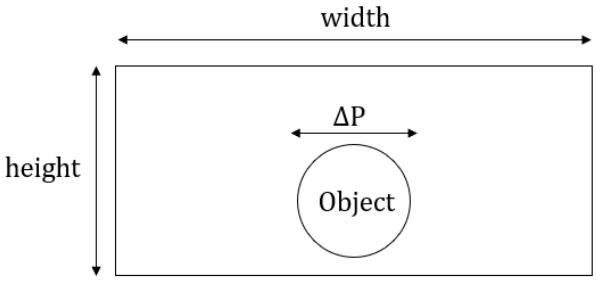


Figure 27: schematic of the picture.

The next figure is a schematic of our picture in real. We assumed the object is in the middle of the picture.

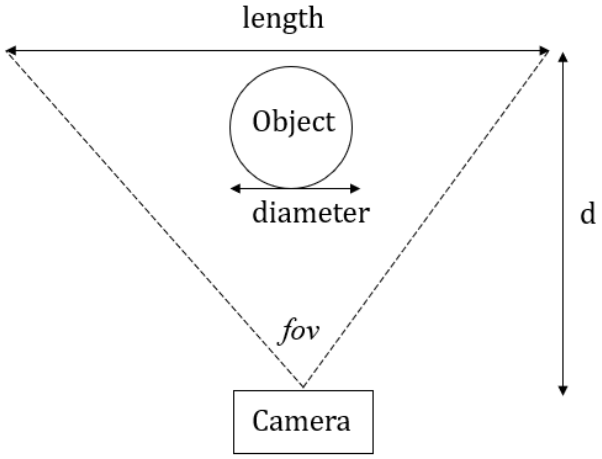


Figure 28: schematic of picture in real.

Now, to calculate the distance d between the object and the camera we assumed that the proportion between width and ΔP on the picture is the same as length and diameter in our world.

$$\frac{width}{\Delta P} = \frac{length}{diameter}$$

We calculated the distance between the object in the centre of the picture and the cameras as the same way as in part 2 by using trigonometry formula. *fov* is the vertically field of view calculated in part 2.

$$\frac{length}{2} = \tan\left(\frac{fov}{2}\right) \times d$$

We can combine the two last equations and isolate d.

$$\frac{width}{\Delta P} = \frac{2\tan\left(\frac{fov}{2}\right) \times d}{diameter}$$

$$d = \frac{width \times diameter}{\Delta P \times 2\tan\left(\frac{fov}{2}\right)}$$

$$d = \frac{921 \times 0.045}{(504 - 392) \times 2\tan\left(\frac{64}{2}\right)}$$

$$d = 0.29\,m$$

When we took the picture, we were 35cm away from the clementine. The result we obtained is close but not precise enough. Maybe it is because the value we uses are too much round or the way we theoretically calculate the distance is wrong.