Image processing
Final project

By:

Ofir Morhaim

205648108

&

Noam Bassat

308465434

Table Of Content

Visible Watermark

In this section, we have created a function that gets as inputs: An image, a watermark image, an intensity value, a height position and a width position and returns the same image with a visible watermark.

```
def add_visible_watermark(img, logo, intensity, w_img, h_img):
    h_logo, w_logo, c_logo = logo.shape
    y_up = h_img + int(h_logo/2)
    x_left = w_img - int(w_logo/2)
    y_down = h_img -int(h_logo/2)
    x_right = w_img + int(w_logo/2)+1

    print("y_down: ", y_down)
    print("y_up: ", y_up)
    print("x_left: ", x_left)
    print("x_right: ", x_right)
    print("shape: ", img.shape)
    img[y_down : y_up, x_left: x_right] = img[y_down : y_up, x_left:
x_right] - (logo-intensity)
    return img
```

Description:

Inputs:

"img" = The main image that we want to add watermark into; "logo" – the watermark image that needs to be added; intensity – how strongly the watermark will appear on the image; 'w_img' and 'h_img' – the width and height positions for adding the watermark on.

Output:

The original image with visible watermark in the desired position.

Functionality:

1. Calculate h_logo, w_logo and c_logo as the height, width and color of the watermark.
2. Set the correct position for pasting the watermark:

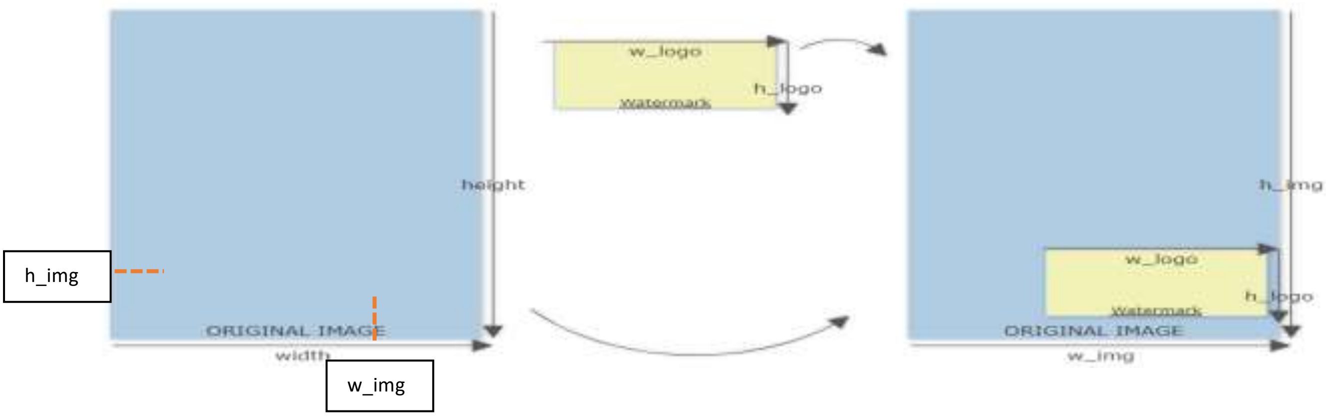
y_up position is calculated by adding to the original image height, half of the watermark height value.

x_left position is calculated by subtracting from the original image width, half of the watermark width value.

y_down position is calculated by subtracting from the original image height, half of the watermark height value.

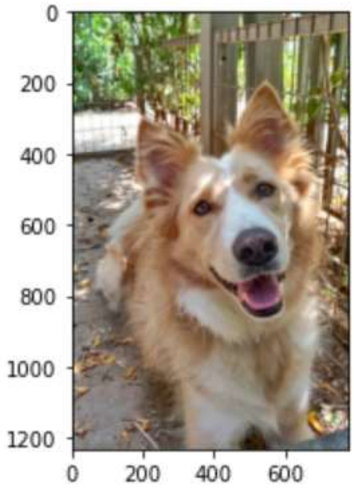
y_right position is calculated by adding to the original image width, half of the watermark width value.

3. Finally, subtract the intensity from the logo, and then subtract the new logo from the positions above, in the original image.



Before:

Original image



Watermark

Results:

img2 = add_visible_watermark(img, watermark, **0.25, 600, 1100**)

Image with visible watermark

```
y_down:  1056
y_up:  1144
x_left:  440
x_right:  761
shape:  (1233, 778, 3)
```

img3 = add_visible_watermark(img, watermark, **0.5, 400, 1100**)

```
y_down:  1056
y_up:  1144
x_left:  240
x_right:  561
shape:  (1233, 778, 3)
```

Invisible Watermark

In this section, we have created a function that gets as inputs: an image and a watermark image, and returns the same image with an invisible watermark.

```
def add_invisible_watermark(img,watermark):
    i=0
    data = "0"
    for x in watermark:
        for y in x:
            for z in y:
                data += str(z)
    with Image.open("Lotus.jpg") as inv_wa_img:
        width, height = inv_wa_img.size
        for x,y,n in itertools.product(range(width), range(height), range(3)):
            pixel = list(inv_wa_img.getpixel((x, y)))
            if(i<len(data)):
                pixel[n] = pixel[n] & ~1 | int(data[i]) # (pixel[n] and not 1) or data
                i+=1
            inv_wa_img.putpixel((x,y), tuple(pixel))
        inv_wa_img.save("source_secret.png", "PNG")
    plt.imshow(inv_wa_img)
    return inv_wa_img
```

Functionality:

1. First, disassemble the watermark, into binary bits data string. Declare an integer 'i'=0 that presents the current index in the data string.
2. Loop over the image's width and height, and make a list if its pixels.
3. For each pixel in the three color channels (RGB), which is a type of UINT8, store the value in the pixel's list at the 'n' position (**pixel[n]**).
4. For each pixel[n]: pixel[n] = pixel[n] AND ~1 OR data[i]:

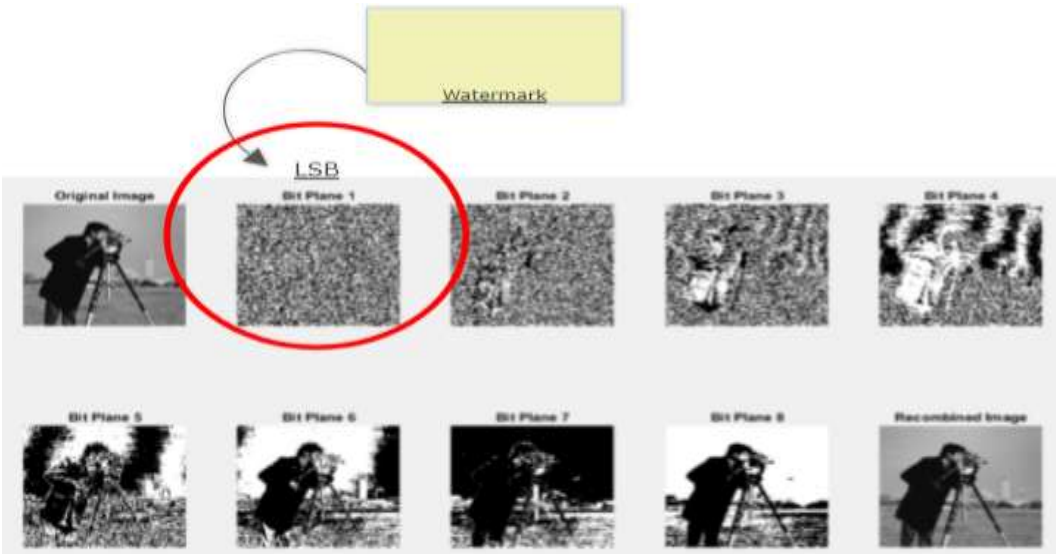
~1 = 1complete = 11111110

(UINT8)X AND ~1 => Reset the value of the LSB. For example:

(DEC)7 =(UINT8)0000111; ~1 = 11111110 -> 0000111 (**AND**) 11111110 = 0000110 = (DEC)6.

In this way, the least significant bit is deleted. Then, adding the "OR" condition to it, so this bit becomes 1 again or stays zero, depending on the data[i] current value.

5. Finally, add the new changed pixels into the original image and return it.



Before:

After:

Check changes on the original image:

In this section, we have created a function that compares the lower pixels of a given image to those of the original one (with the invisible watermark).

```
def check_changes(im2):
    with Image.open("source_secret.png") as im1:
            width, height = im1.size
            if (im1.size!= im2.size): return "Different images!"
            for x,y in itertools.product(range(width),
range(height)):
               pixel = list(im1.getpixel((x, y)))
               pixel2 =list(im2.getpixel((x, y)))
               for n in range(0,3):
                       if(pixel[n]!=pixel2[n]): return "Changed!!!"
    return "Pure Image!"
```

Description:

Inputs:

"img2" = The same image as the original one, with a minor filter added.

Output:

A string that indicates whether the original image has been changed or not.

Functionality:

1. If the images sizes are different, returns "Different images!"
2. Else, looping over the images height and width and storing the original image's pixels into pixel list, and the other image into pixel2 list.
3. For each pixel in the three color channels (RGB), which is a type of UINT8, store the value in the pixel's list at the 'n' position (**pixel[n]** for the original image and **pixel2[n]** for the other image.)
4. Compare each pixel with the parallel pixel2, if they are not equal, return "Changed!!!"
5. Else, continue until the end, if there were not any different pixels, return "Pure Image".

```
print("equal = ",check_changes(inv_wa_img)) #  same image
```

```
equal =  Pure Image!
```

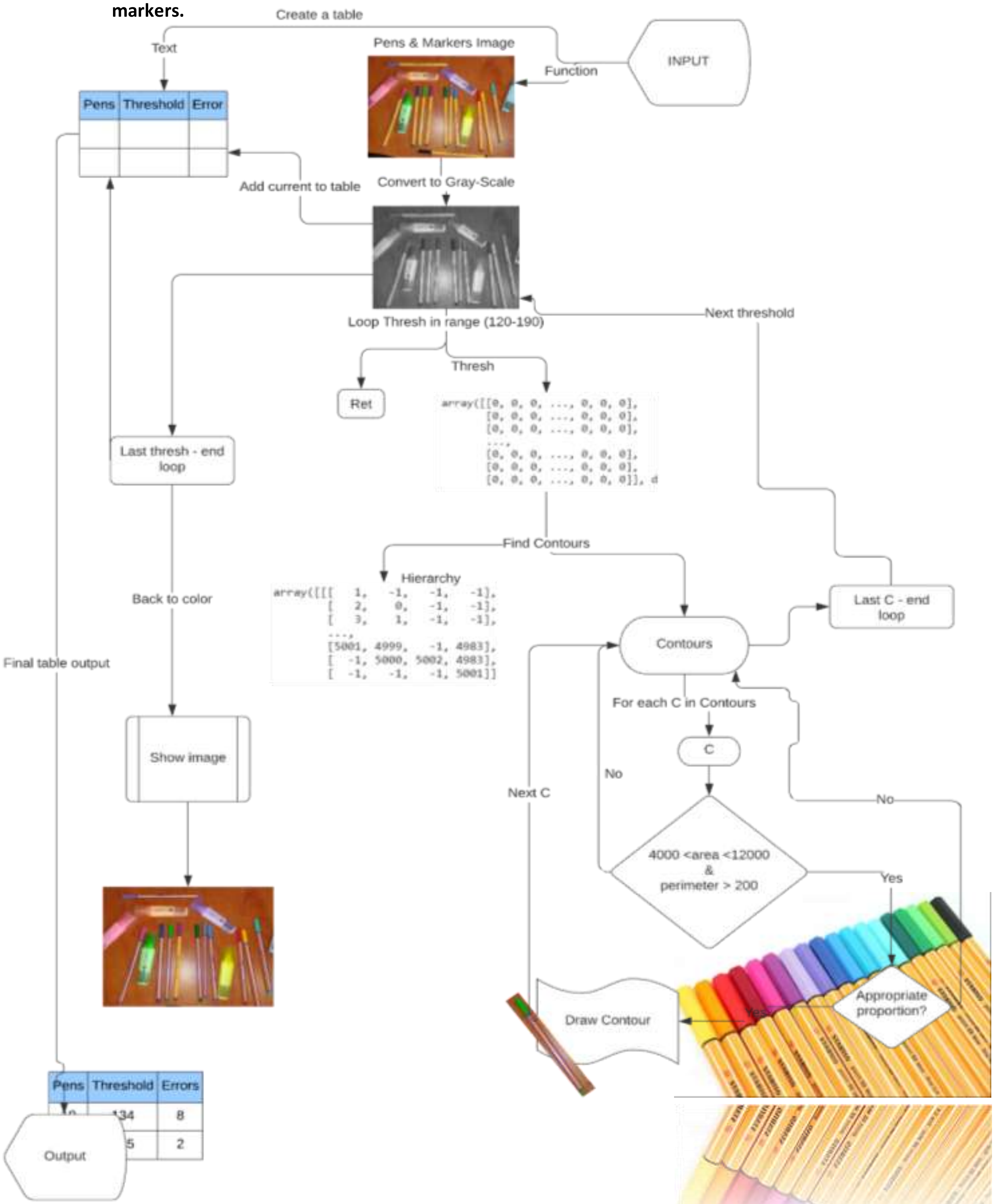
```
print("equal = ",check_changes(IMAGE)) # same image after a tiny filter
```

```
equal =  Changed!!!
```

A system for counting and identifying objects from the same category

In this section, we have created a system for identifying and counting **pens**, in a given image.

In order to check the system's performance, we sent images that showed both **pens and markers.**



The code:

```
def obObject_in_img(image):
    df = pd.DataFrame(index=range(11),
columns=["Threshold","Num_of_Pens_Found","FP","FN"])
    threshold = 115 # Initialize the threshold
    print("Original thresh = ",threshold_otsu(image))
    for i in range(11):
        image_c = image.copy()
        threshold += 7
        print("threshold = ", threshold)
        df["Threshold"][i] = threshold
        img_gray = cv2.cvtColor(image_c, cv2.COLOR_BGR2GRAY) #
Convert a color rgb image to a grayscale image
        ret, thresh = cv2.threshold(img_gray, threshold,255,
cv2.THRESH_BINARY) # Use a binary threshold to find a large contrast
        contours, hierarchy = cv2.findContours(image=thresh,
mode=cv2.RETR_TREE, method=cv2.CHAIN_APPROX_NONE)
        count=0
        for c in contours:
            area = cv2.contourArea(c)
            perimeter = cv2.arcLength(c,True)
            x,y,w,h = cv2.boundingRect(c)
            if area>4000 and perimeter >200 and area<12000: # Range
of possible areas for the pen (depending on whether it is placed
diagonally)
                rect = cv2.minAreaRect(c) # Rect = Min area
                box = cv2.boxPoints(rect) # Box = Slope of objects
placed diagonally
                box = np.int0(box)
                if ( h>400 and w<100*rect[2] ) or ( w>400 and
h<100*rect[2] ):  # Conditions for pen objects
                    cv2.drawContours(image_c,[box],0,(255,0,50),3) #
Drawing the Contours
                    count += 1
        df["Num_of_Pens_Found"][i] = count
        img1 = cv2.cvtColor(image_c , cv2.COLOR_BGR2RGB) # Convert
the Grayscale image back to RGB
        plt.figure(figsize =(8,9))
        plt.imshow(img1)
        plt.show()
    return df
```

Description:

Inputs:

"Image" = Image of pens and markers.

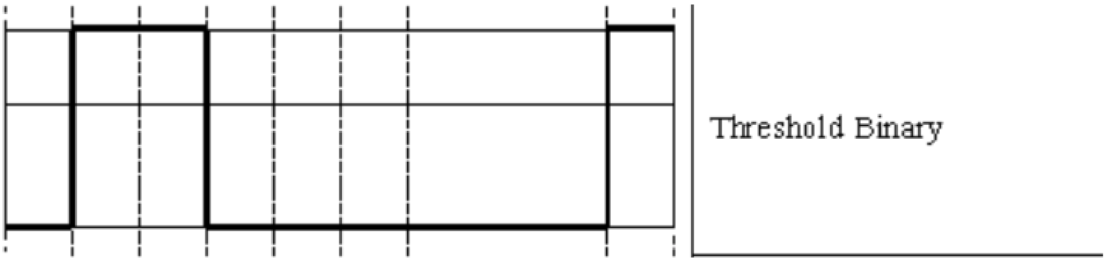
Output:

A Data-Frame object which represents a table of all threshold values and the number of pens that were identified for each threshold value.

Functionality:

1. Create data frame and set the threshold value to 115.

2. Loop over the threshold range: 115-192, with steps of 7. (AKA 115, 122, 129 … , 192).
3. Convert the original RGB image into a grayscale image.
4. Set ‘ret’, ‘thresh’ = a **<u>binary threshold:</u>** If pixel intensity is greater than the set threshold, value set to 255, else set to 0 (black). [4]



$$\mathtt{dst}(x,y) = \begin{cases} \mathtt{maxval} & \text{if } \mathtt{src}(x,y) > \mathtt{thresh} \\ 0 & \text{otherwise} \end{cases}$$

[1]

5. Set ‘contour’, ‘hierarchy’ = the cv2.findContours function. In this method all the boundaries points are stored.
   Actually, we do not need all the points. In order to find the contour of a straight line all we need is just two ends of that line. This is what cv.CHAIN_APPROX_SIMPLE parameter does. It removes all redundant points and compresses the contour, thereby saving memory. [2]

   <u>‘Contours’</u> is a list, or tree of lists of points. The points describe each contour, that is, a vector that could be drawn as an outline around the parts of the shape based on their difference from a background.

   <u>‘Hierarchy</u>’ shows how the shapes relate to each other, layers as such - if shapes are on top of each other this can be determined here. [3]

6. Loop over each ‘c’ in <u>Contours’.</u>
7. Use the green formula to calculate the area of contour and set it as ‘area’:

$$\text{Area of } D = \iint_D dA = \iint_D 1\, dA.$$

[5]

8. Calculate a contour perimeter or a curve length and set it as ‘perimeter’
9. Set ‘x’, ’y’ as the object’s coordinates in the image, while ‘w’ and ‘h’ are the width and the height of the object.
10. For each recognized object, check its area size conditions. If it matches the conditions, (means we found a pen), draw a rectangle around it.
11. Finally, import the gray-scaled with the rectangles image back to RGB color image, present it, and return the table of values.

The images:

We tested different images with different light conditions and angles:

(1) No – flash *Best performance: (threshold = 150, FN = 4)*

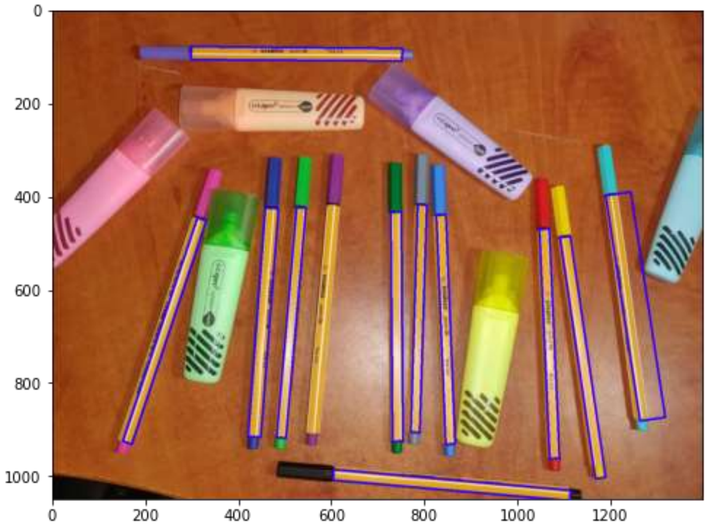
(2) With Flash *Best performance: (threshold = 150, FN = 1)*

(3) No flash, different angel

Best performance: (threshold = 164, FN = 3)

***(4)** With flash different angle*

Best performance: (threshold = 157, FN = 1)

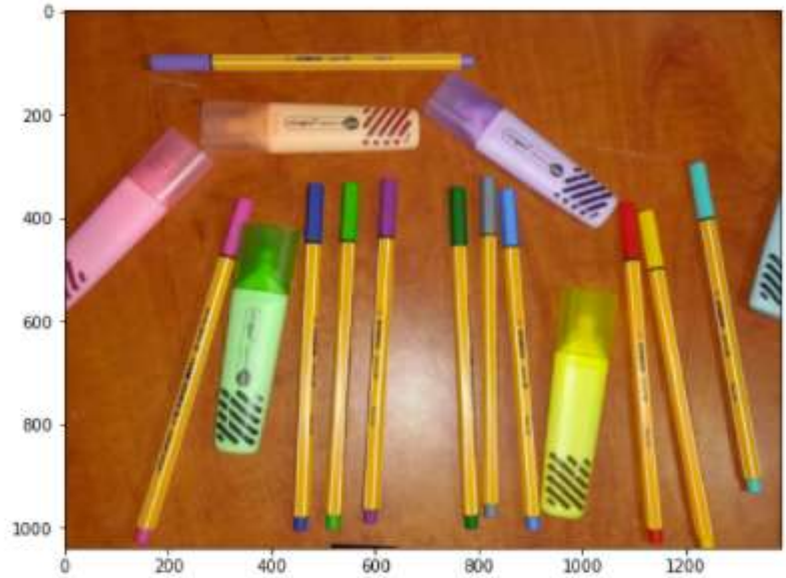


Conclusion:

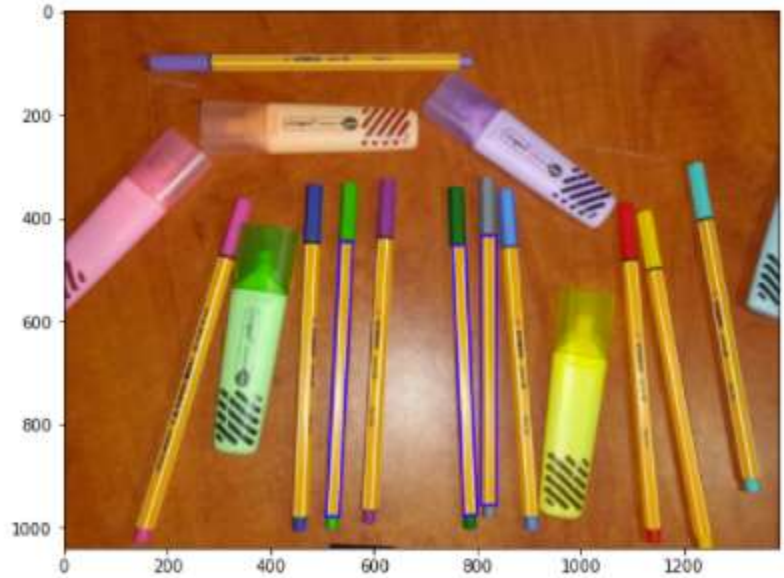
The best conditions for the pens detection is the images that were taken with flash.

Threshold process over an image:

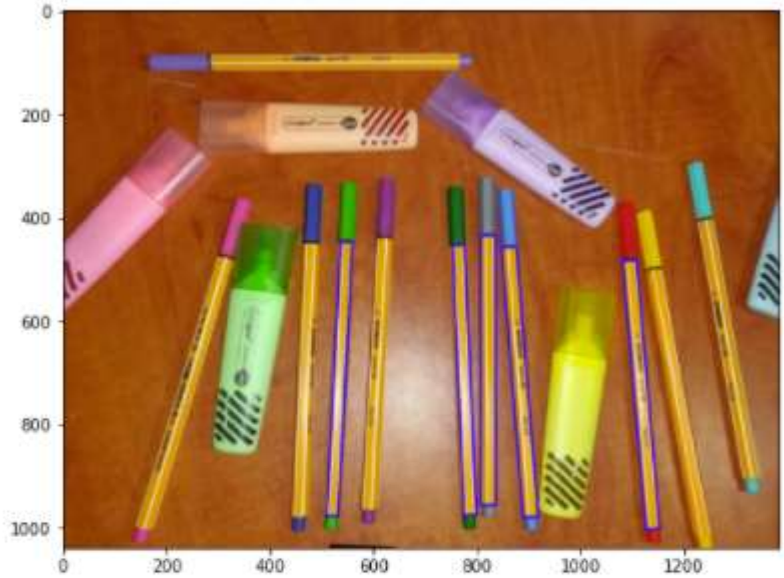
threshold = 129



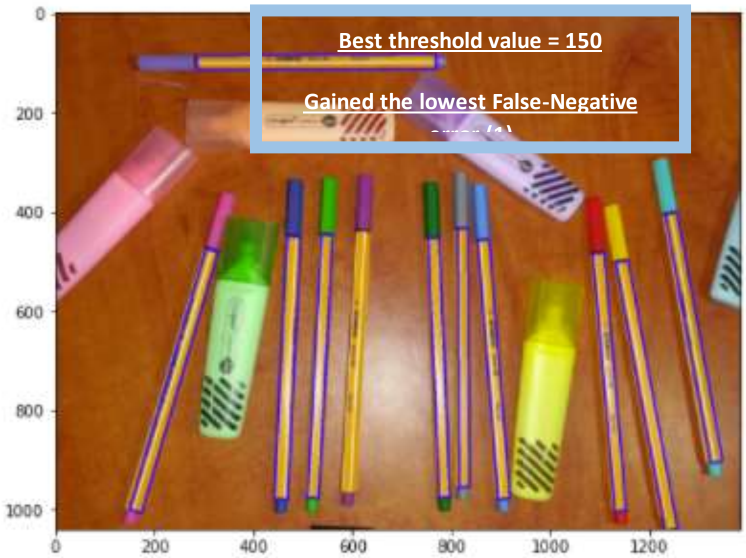
threshold = 136



threshold = 143



threshold = 150



threshold = 157

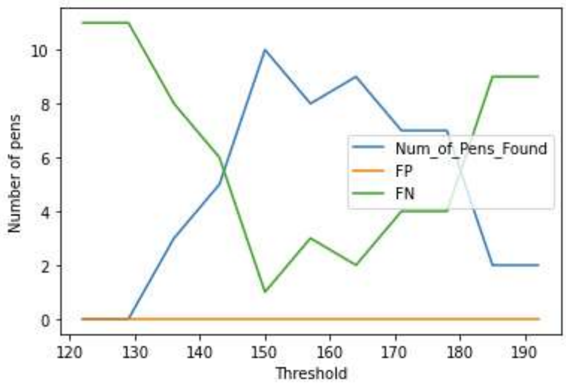
threshold = 164

Threshold Graphs

Present all performances – after we adjusted the model, no false positive errors performed.

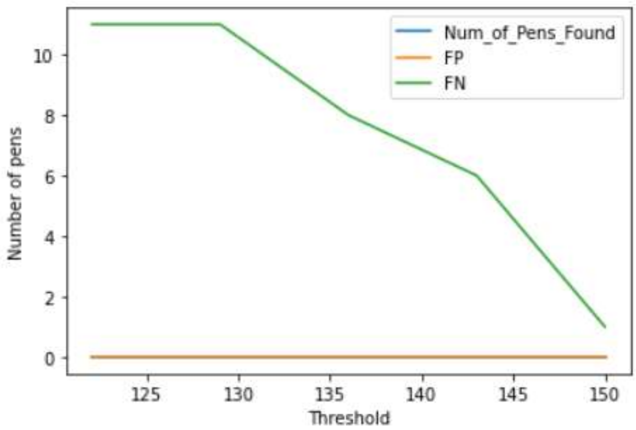
Threshold range: 122-192

	Threshold	Num_of_Pens_Found	FP	FN
0	122	0	0	11
1	129	0	0	11
2	136	3	0	8
3	143	5	0	6
4	150	10	0	1
5	157	8	0	3
6	164	9	0	2
7	171	7	0	4
8	178	7	0	4
9	185	2	0	9
10	192	2	0	9



Threshold range: 122-150

	Threshold	Num_of_Pens_Found	FP	FN
0	122	0	0	11
1	129	0	0	11
2	136	0	0	8
3	143	0	0	6
4	150	0	0	1



Conclusions:

The best images for this mission were those taken with flash.

Best threshold value is 150.

The correct area for a pen must be in the range of 4000-12000 (depending on the angle position of the pen), while the circumference must be greater than 200.

Auxiliary sources

[1]
https://docs.opencv.org/4.5.2/d7/d1b/group__imgproc__misc.html#ggaa9e58d2860d4afa658ef70a9b1115576a147222a96556ebc1d948b372bcd7ac59

[2] https://docs.opencv.org/4.5.2/d4/d73/tutorial_py_contours_begin.html

[3] https://stackoverflow.com/questions/46236061/what-are-values-returned-by-findcontour-function-in-opencv-2-4-9, answer by Danny Staple.

[4] https://www.geeksforgeeks.org/python-thresholding-techniques-using-opencv-set-1-simple-thresholding/

[5]
https://mathinsight.org/greens_theorem_find_area#:%7E:text=We%20can%20parametrized%20it%20in,dt%3D%CF%80r2