Assignment 2 Report

Digital Image Processing

By:

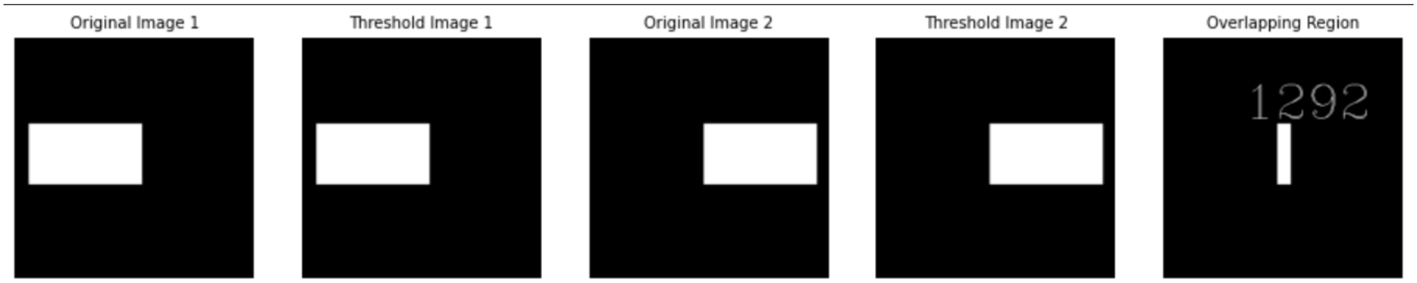
Danyal Faheem

19I-2014 CS-F

Question 1:

First, I read the images using their respective paths. Then I applied thresholding on both images to convert them into binary images. After they had been converted into binary images, I applied the Bitwise and operator on the images using the cv2.bitwise_and() function in OpenCV. This allowed me to get a new image with just the intersection of the other two images. Finally, I had applied the cv2.countNonZero() function to get the number of white pixels on the image which would be the area of the intersection. Then I used the cv2.putText() function to display the calculated area onto the image.

Output:



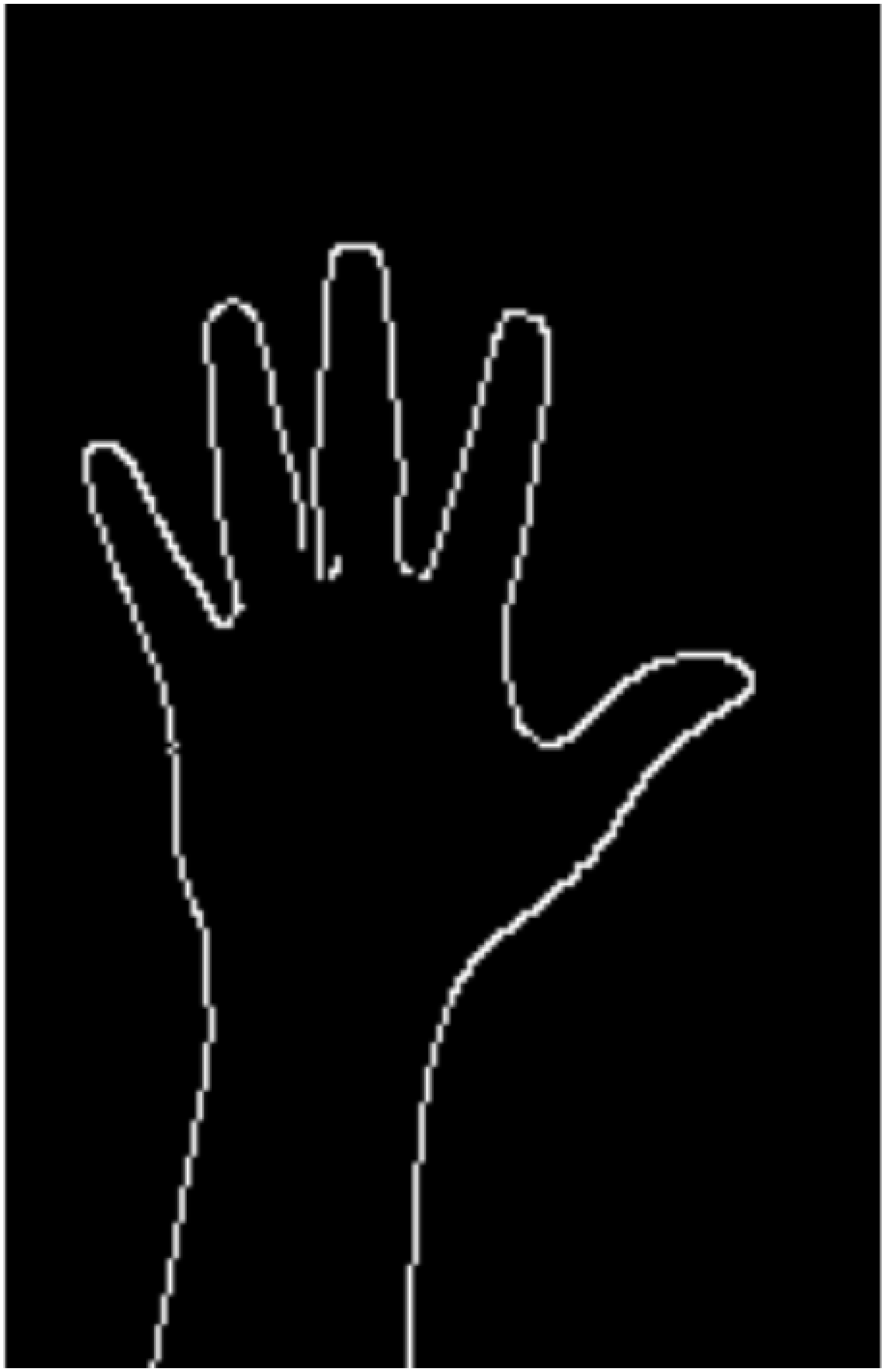
Snippet of code:

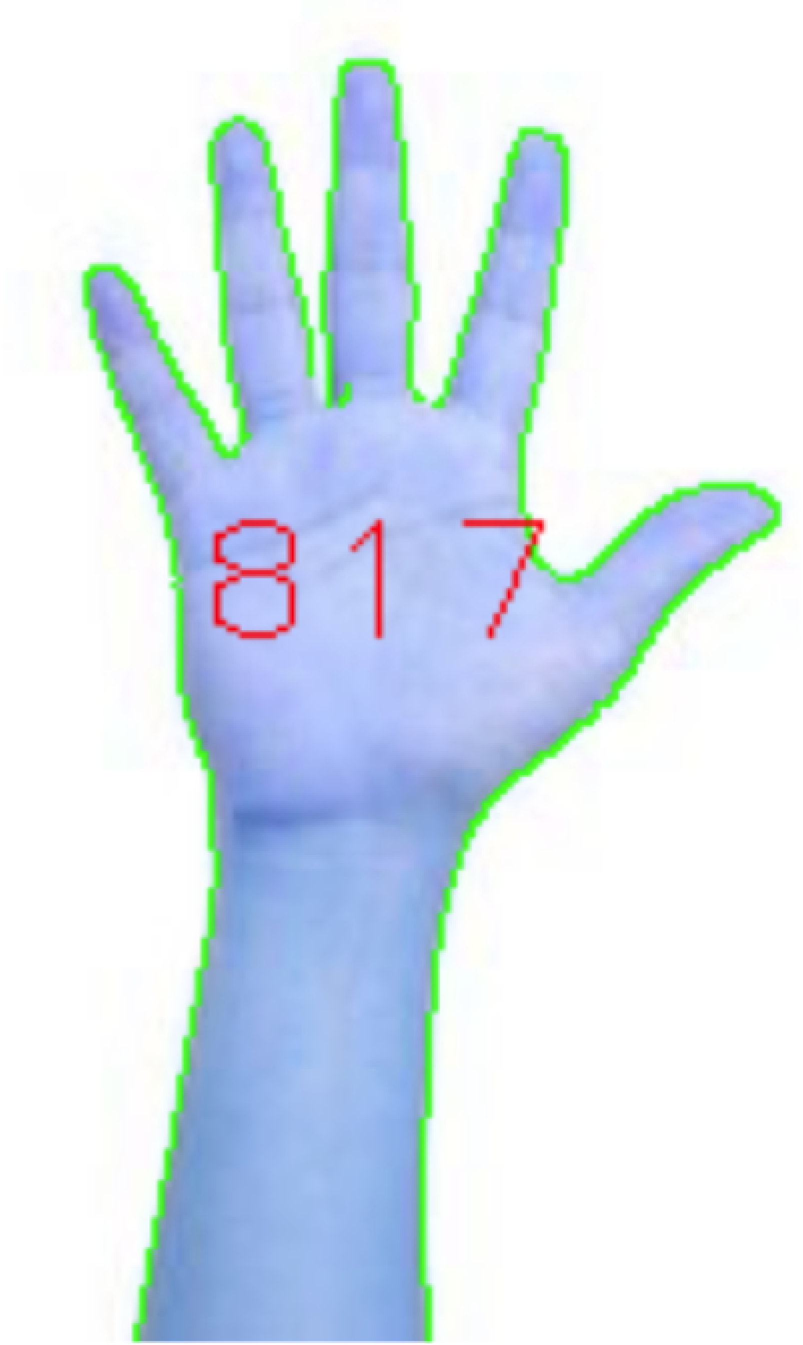
```
#Thresholding images to convert into a binary image

threshold, imga1 = cv2.threshold(imga, 80, 255, cv2.THRESH_BINARY)

threshold, imgb1 = cv2.threshold(imgb, 80, 255, cv2.THRESH_BINARY)


#Applying bitwise_and to get the intersection of the 2 images

img_bwa = cv2.bitwise_and(imga1, imgb1)


#Displaying the values number of non zero pixels on the image

cv2.putText(img_bwa,str(cv2.countNonZero(img_bwa)), (100,100), 3, 2, 255)
```

Question 2:

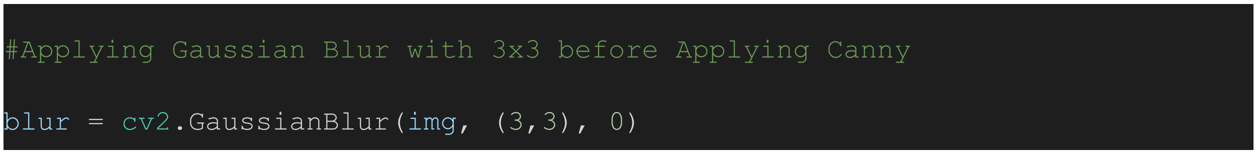
Since I know that there is only one object on the image, I had used the canny edge detector on the image to get singular pixel edges. This gave me a line of edges of the hand. Using these pixel points, I traversed the new image with just the edges highlighted and wherever I found a white pixel, I would change the colour of the x and y coordinates to green in the original image. This resulted in a green line around the edges of the hand in the image. Afterwards, I counted the number of white pixels using the cv2.countNonZero() in the canny edge image which gave singular pixels for the edge and this gave me the perimeter of the hand which I displayed on the original image using the cv2.putText() function.

Outputs:





Snippet of code:

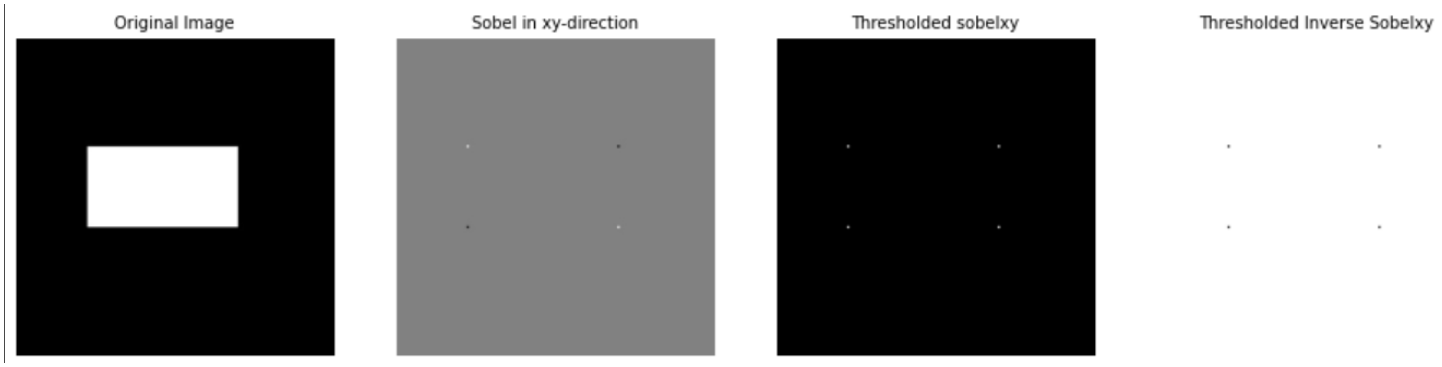


```
#Applying Gaussian Blur with 3x3 before Applying Canny

blur = cv2.GaussianBlur(img, (3,3), 0)
```

```python
# Applying Canny with and without Gaussian blur to check effect
edgesCanny = cv2.Canny(img, threshold1=100, threshold2=200)
edgesCanny1 = cv2.Canny(blur, threshold1=110, threshold2=180)

#Thresholding the image to convert into binary
threshold, imga1 = cv2.threshold(edgesCanny1, 80, 255, cv2.THRESH_BINARY)

#Function to make pixel green on edges
def draw():
    counter = 0
    for x in imga1:
        count = 0
        for y in x:
            if y > 200:
                #If pixel is white in canny edges, change colour to green to signify edge
                img_colour[counter][count] = (0,255,0)
            count += 1
        counter += 1
draw()

#Writing the count of edge pixels as they are perimeter onto image
cv2.putText(img_colour,str(cv2.countNonZero(imga1)), (30,140), 1, 2, 255)
```

Question 3:

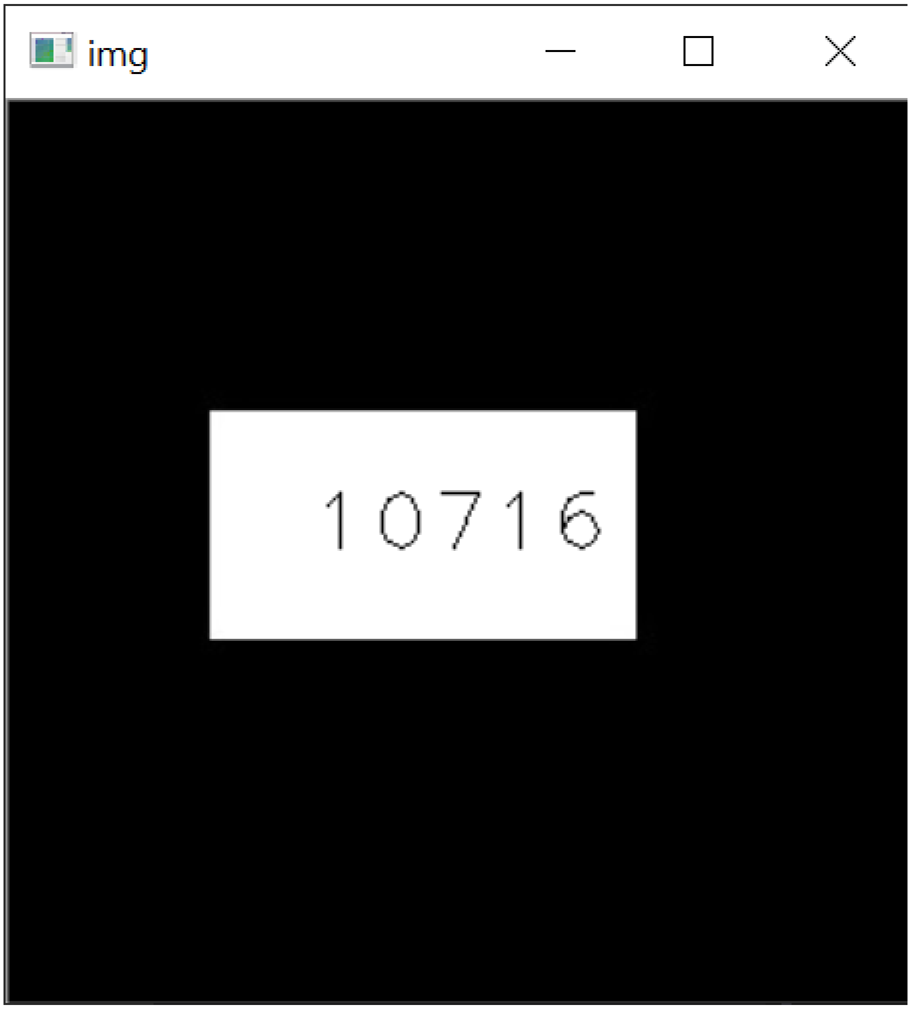
For this part, I had used the sobel operator in the xy direction using the inbuilt Sobel function in OpenCV. Due to both xy directions, the corners were highlighted and the rest discarded. After this, I took the absolute value of the sobel xy image. Thereafter, I thresholded the image to get a binary image. This gave me 4 points which were 4 corners of the object in the image.



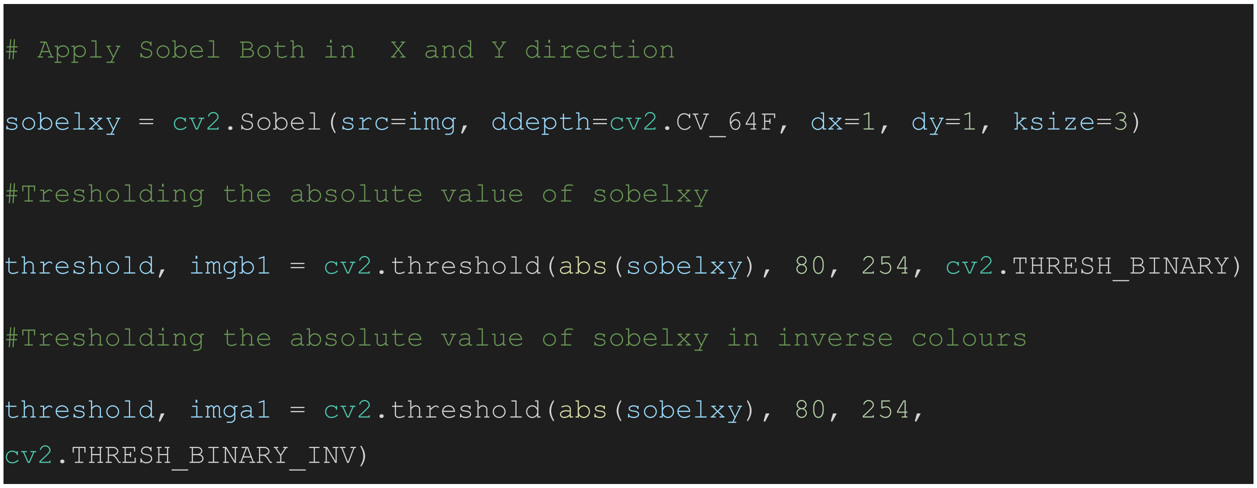
Once I had the corners, I used a nested for loop to traverse the image and as soon as I got a white pixel, I would use another loop while keeping the x coordinate constant and only changing the y coordinate to get the width of the object. I also kept the y coordinate constant in another loop while changing the x coordinate to get the length of the object. I ran these loops till a white pixel was found.

Thereafter, I used the resulting length and width and some past knowledge to see that it was a rectangular shape. Using the formula area = length * width, I calculated the area of the object and displayed it on the image using the cv2.putText() function.

Output:



Snippet of code:



```
# Apply Sobel Both in  X and Y direction

sobelxy = cv2.Sobel(src=img, ddepth=cv2.CV_64F, dx=1, dy=1, ksize=3)

#Tresholding the absolute value of sobelxy

threshold, imgb1 = cv2.threshold(abs(sobelxy), 80, 254, cv2.THRESH_BINARY)

#Tresholding the absolute value of sobelxy in inverse colours

threshold, imga1 = cv2.threshold(abs(sobelxy), 80, 254,
cv2.THRESH_BINARY_INV)
```

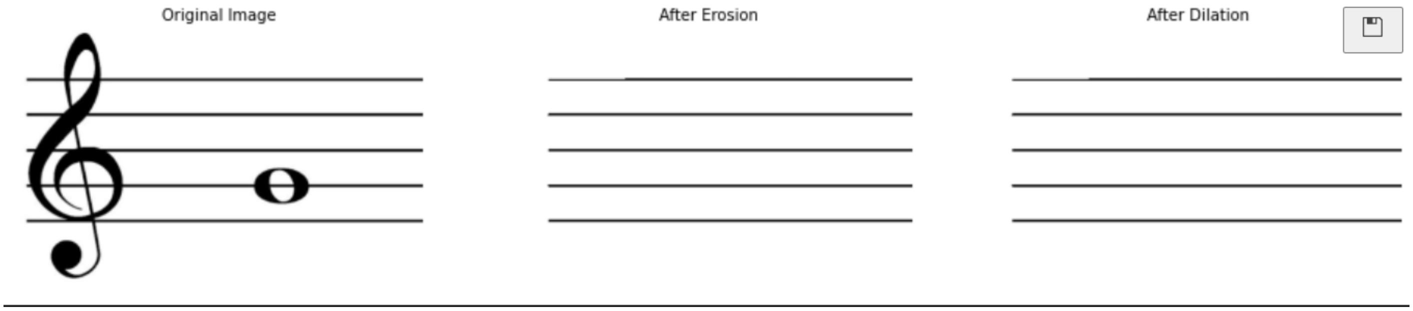
```
#Defining global variables
counter = 0
length = 0
width = 0

#Function to calculate the length and width
def calcSides():
    global counter, length, width
    for x in imgb1:
        count = 0
        for y in x:
            #if pixel is white
            if y > 200:
                index_x = counter + 20
                index_y = count + 10
                #Loop to increment length till next pixel found on the same x axis
                while True:
                    if imgb1[index_x][count] > 200:
                        break
                    length += 1
                    index_x += 1
                #Loop to increment width till next pixel found on the same y axis
                while True:
                    if imgb1[counter][index_y + 1] > 200:
                        break
                    width += 1
                    index_y += 1
                #return to stop infinite looping
                return
            count += 1
        counter += 1
calcSides()

#Writing the area of rectangle on the image
cv2.putText(img, str(length * width), (100, 150), 1, 2, 0)
```

Question 4:

For this problem, I had used the concept of morphological processing for a solution. I had first thresholded the image to get a binary image. Thereafter, I had taken the negative of the thresholded image so we can apply morphological processing in order.

After this, I created a kernel of size 1x10 to erode and dilate the image. I had then applied erosion with the kernel using the cv2.erode() in OpenCV. I had run 8 iterations as that would give me the correct result. Then, I had dilated the image with the same kernel and 8 iterations so that any horizontal lines that were eroded could be brought back. Then, I took the negative of the final image to be able to display it again in original format.

Output:



Code snippet:

```
#Thresholding image to convert to binary

threshold, img1 = cv2.threshold(img, 80, 254, cv2.THRESH_BINARY)

#Taking the negative of image to apply morphological processing

img1 = 255 - img1

#Creating our kernel for the Dilate and Erode operations

kernel = np.ones((1,10),np.uint8)

#Applying erosion on the image to extract only horizontal lines

erosion = cv2.erode(img1,kernel,iterations = 8)

#Applying dilation on the image to get back the same length of horizontal
lines

dilation = cv2.dilate(erosion,kernel,iterations = 8)

#Again taking the negative to display as original image
```

```
dilation = 255 - dilation

erosion = 255 - erosion
```

Thank You!