DIGITAL IMAGE PROCESSING COURSE - 505060
PRACTICE LABS

LAB 04. MORPHOLOGY IMAGE PROCESSING

Requirements

(1) Follow the instructions with the help from your instructor.
(2) Finish all the exercises in class and do the homework at home. You can update your solutions after class and re-submit all your work together with the homework.
(3) Grading
Total score = 50% * Attendance + 50% * Exercises
Rules:
- If the number of finished exercises is less than **80% total number of excercises**, you will get **zero** for the lab.
- Name a source file as "**src_XX.py**" where XX is the exercise number, for ex., "src_03.py" is the source code for the Exercise 3.
- Add the text of your Student ID to each of the output image.
- Name an output image as "**image_XX_YY.png**" where XX is the exercise number and YY is the order of output images in the exercise, for ex., "image_03_02.png" is the second output image in the Exercise 3.
- Submit the source code and output image files directly to Google classroom assignment, donot compress the files.

If you submit the exercises with wrong rules, you will get **zero** for the lab or the corresponding exercises.

(4) Plagiarism check
If any 2 of the students have the same output images, then all will get zero for the corresponding exercises.

INTRODUCTION

In this Lab, you will learn how to
- You will learn Simple thresholding, Adaptive thresholding, Otsu's thresholding etc.
- You will learn these functions : cv2.threshold, cv2.adaptiveThreshold etc.
- We will learn different morphological operations like Erosion, Dilation, Opening, Closing etc.
- We will see different functions like : cv2.erode(), cv2.dilate(), cv2.morphologyEx() etc.
- Contour in OpenCV

INSTRUCTIONS

---

1. Image thresholding

https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html

Adaptive Thresholding

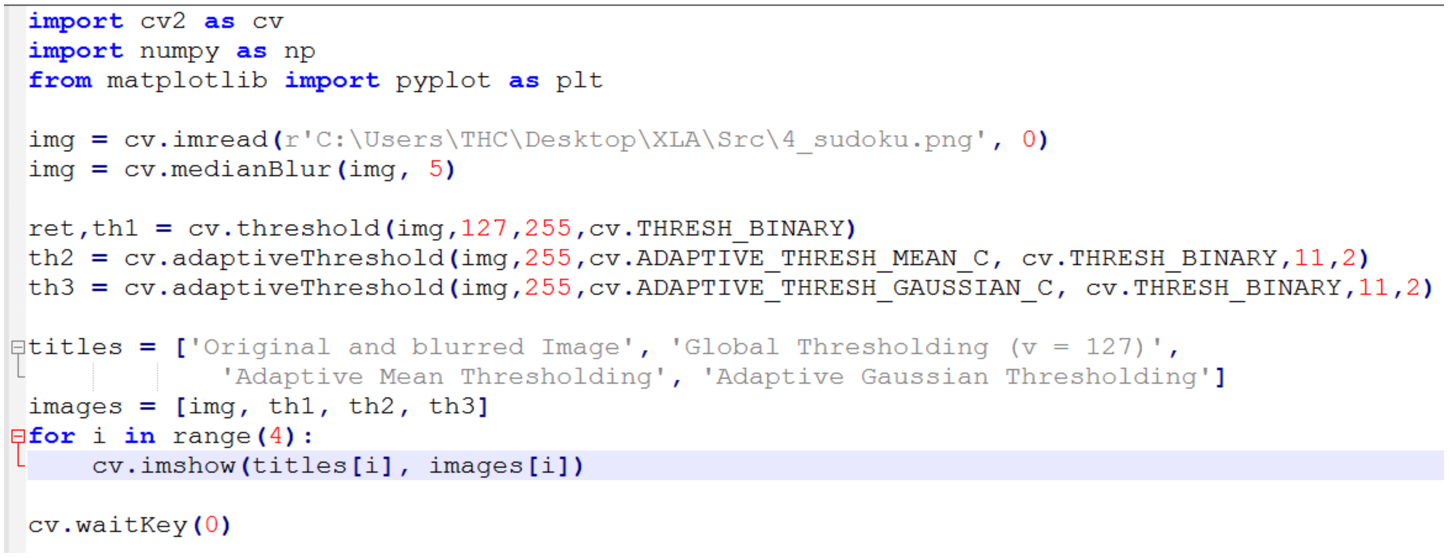
In the previous section, we used one global value as a threshold. But this might not be good in all cases, e.g. if an image has different lighting conditions in different areas. In that case, adaptive thresholding can help. Here, the algorithm determines the threshold for a pixel based on a small region around it. So we get different thresholds for different regions of the same image which gives better results for images with varying illumination.

```
dst = cv.AdaptiveThreshold(src, maxValue,
adaptive_method=CV_ADAPTIVE_THRESH_MEAN_C, thresholdType=CV_THRESH_BINARY,
blockSize, C)
```

where:

0. **dst** - Destination image of the same size and the same type as **src**.
1. **src** - Source 8-bit single-channel image.
2. **maxValue** - Non-zero value assigned to the pixels for which the condition is satisfied.
3. **adaptiveMethod** decides how the threshold value is calculated:
    a. **ADAPTIVE_THRESH_MEAN_C** : The threshold value is the mean of the neighbourhood area minus the constant **C**.
    b. **ADAPTIVE_THRESH_GAUSSIAN_C** : The threshold value is a gaussian-weighted sum of the neighbourhood values minus the constant **C**.
4. **thresholdType** - Thresholding type that must be either THRESH_BINARY or THRESH_BINARY_INV .
5. **blockSize** determines the size of the neighbourhood area.
6. **C** - Constant subtracted from the mean or weighted mean. Normally, it is positive but may be zero or negative as well. It is just a constant which is subtracted from the mean or weighted mean calculated.

Example:

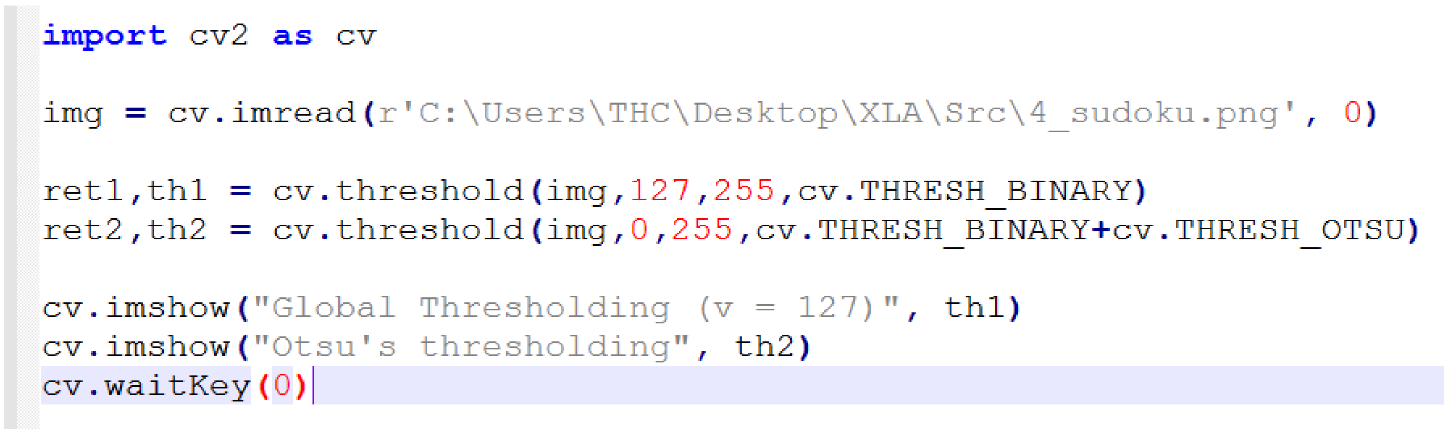


```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt

img = cv.imread(r'C:\Users\THC\Desktop\XLA\Src\4_sudoku.png', 0)
img = cv.medianBlur(img, 5)

ret,th1 = cv.threshold(img,127,255,cv.THRESH_BINARY)
th2 = cv.adaptiveThreshold(img,255,cv.ADAPTIVE_THRESH_MEAN_C, cv.THRESH_BINARY,11,2)
th3 = cv.adaptiveThreshold(img,255,cv.ADAPTIVE_THRESH_GAUSSIAN_C, cv.THRESH_BINARY,11,2)

titles = ['Original and blurred Image', 'Global Thresholding (v = 127)',
            'Adaptive Mean Thresholding', 'Adaptive Gaussian Thresholding']
images = [img, th1, th2, th3]
for i in range(4):
    cv.imshow(titles[i], images[i])

cv.waitKey(0)
```

Otsu Threshold (Separating Bimodal Distributions)

Otsu binarization automatically calculates a threshold value from image histogram for a bimodal image. It uses **cv2.threshold()** function with an extra flag, **cv2.THRESH_OTSU**. For threshold value, simply pass zero. Then the algorithm finds the optimal threshold value which is returned as the first output.

Example:



```
import cv2 as cv

img = cv.imread(r'C:\Users\THC\Desktop\XLA\Src\4_sudoku.png', 0)

ret1,th1 = cv.threshold(img,127,255,cv.THRESH_BINARY)
ret2,th2 = cv.threshold(img,0,255,cv.THRESH_BINARY+cv.THRESH_OTSU)

cv.imshow("Global Thresholding (v = 127)", th1)
cv.imshow("Otsu's thresholding", th2)
cv.waitKey(0)
```

2. Morphology Operations

Structuring Element

We could create the structuring elements with help of Numpy. It has a rectangular shape. But in some cases, you may need elliptical/circular shaped kernels. So for this purpose, we use OpenCV **cv2.getStructuringElement()**.

Source: https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_morphological_ops/py_morphological_ops.html

```
# Rectangular Kernel
>>> cv2.getStructuringElement(cv2.MORPH_RECT,(5,5))
array([[1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1]], dtype=uint8)
```

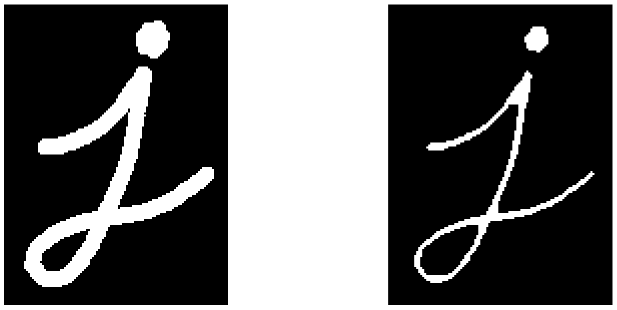
```
# Elliptical Kernel
>>> cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(5,5))
array([[0, 0, 1, 0, 0],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [0, 0, 1, 0, 0]], dtype=uint8)


# Cross-shaped Kernel
>>> cv2.getStructuringElement(cv2.MORPH_CROSS,(5,5))
array([[0, 0, 1, 0, 0],
       [0, 0, 1, 0, 0],
       [1, 1, 1, 1, 1],
       [0, 0, 1, 0, 0],
       [0, 0, 1, 0, 0]], dtype=uint8)
```

Erosion

The basic idea of erosion is just like soil erosion only, it erodes away the boundaries of foreground object (Always try to keep foreground in white). So what does it do? The kernel slides through the image (as in 2D convolution). A pixel in the original image (either 1 or 0) will be considered 1 only if all the pixels under the kernel is 1, otherwise it is eroded (made to zero). So what happends is that, all the pixels near boundary will be discarded depending upon the size of kernel. So the thickness or size of the foreground object decreases or simply white region decreases in the image. It is useful for removing small white noises (as we have seen in colorspace chapter), detach two connected objects etc.

Here, as an example, I would use a 5x5 kernel with full of ones. Let's see it how it works:

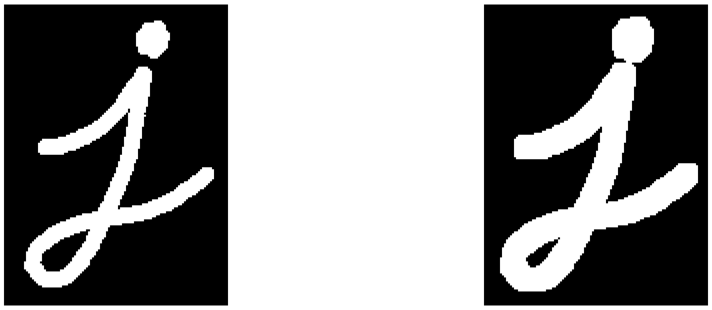
```
import cv2
import numpy as np

img = cv2.imread('j.png',0)
kernel = np.ones((5,5),np.uint8)
erosion = cv2.erode(img,kernel,iterations = 1)
```



Dilation

It is just opposite of erosion. Here, a pixel element is ‘1’ if at least one pixel under the kernel is ‘1’. So it increases the white region in the image or size of foreground object increases. Normally, in cases like noise removal, erosion is followed by dilation. Because, erosion removes white noises, but it also shrinks our object. So we dilate it. Since noise is gone, they won’t come back, but our object area increases. It is also useful in joining broken parts of an object.

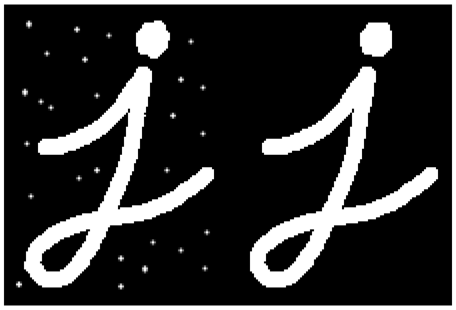
```
dilation = cv2.dilate(img,kernel,iterations = 1)
```



Opening

Opening is just another name of **erosion followed by dilation**. It is useful in removing noise, as we explained above. Here we use the function, **cv2.morphologyEx()**

```
opening = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel)
```



Closing

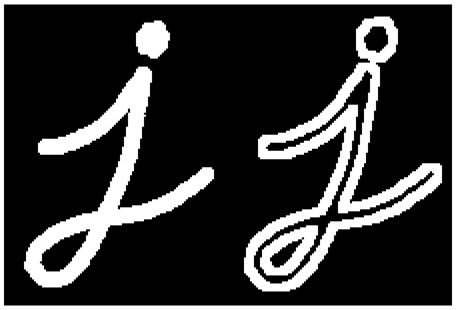
Closing is reverse of Opening, **Dilation followed by Erosion**. It is useful in closing small holes inside the foreground objects, or small black points on the object.

```
closing = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel)
```

Morphological Gradient

It is the difference between dilation and erosion of an image. The result will look like the outline of the object.

```
gradient = cv2.morphologyEx(img, cv2.MORPH_GRADIENT, kernel)
```



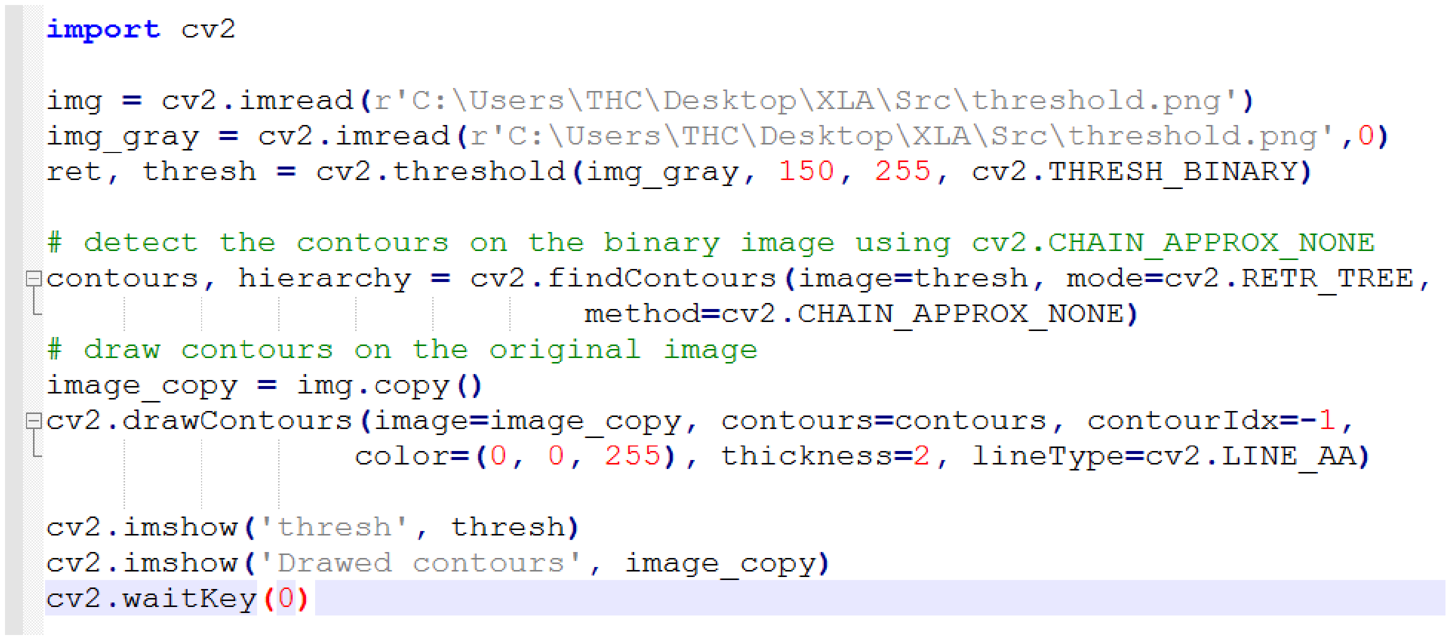
3. Contour Detection using OpenCV

Using contour detection, we can detect the borders of objects, and localize them easily in an image. It is often the first step for many interesting applications, such as image-foreground extraction, simple-image segmentation, detection and recognition.
OpenCV makes it really easy to find and draw contours in images. It provides two simple functions:

- findContours()
- drawContours()

For more information: https://learnopencv.com/contour-detection-using-opencv-python-c/

We specify the method for findContours() as CHAIN_APPROX_SIMPLE or CHAIN_APPROX_NONE.



```
import cv2

img = cv2.imread(r'C:\Users\THC\Desktop\XLA\Src\threshold.png')
img_gray = cv2.imread(r'C:\Users\THC\Desktop\XLA\Src\threshold.png',0)
ret, thresh = cv2.threshold(img_gray, 150, 255, cv2.THRESH_BINARY)

# detect the contours on the binary image using cv2.CHAIN_APPROX_NONE
contours, hierarchy = cv2.findContours(image=thresh, mode=cv2.RETR_TREE,
                                            method=cv2.CHAIN_APPROX_NONE)
# draw contours on the original image
image_copy = img.copy()
cv2.drawContours(image=image_copy, contours=contours, contourIdx=-1,
                color=(0, 0, 255), thickness=2, lineType=cv2.LINE_AA)

cv2.imshow('thresh', thresh)
cv2.imshow('Drawed contours', image_copy)
cv2.waitKey(0)
```

The CHAIN_APPROX_SIMPLE algorithm compresses horizontal, vertical, and diagonal segments along the contour and leaves only their end points. This means that any of the points along the straight paths will be dismissed, and we will be left with only the end points. For example, consider a contour, along a rectangle. All the contour points, except the four corner points will be dismissed. This method is faster than the CHAIN_APPROX_NONE because the algorithm does not store all the points, uses less memory, and therefore, takes less time to execute.

4. Contour Features

To find the different features of contours, like area, perimeter, centroid, bounding box etc

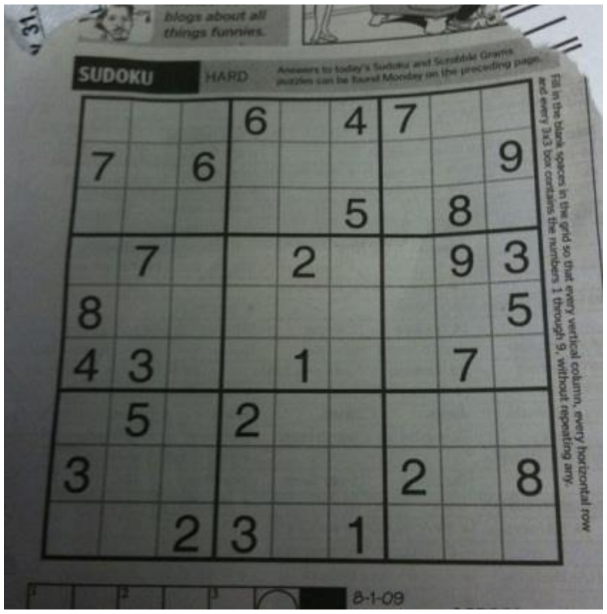
You will see plenty of functions related to contours.

Reference:
https://docs.opencv.org/4.5.3/dd/d49/tutorial_py_contour_features.html

EXERCISES

Ex4.1. Image binarization

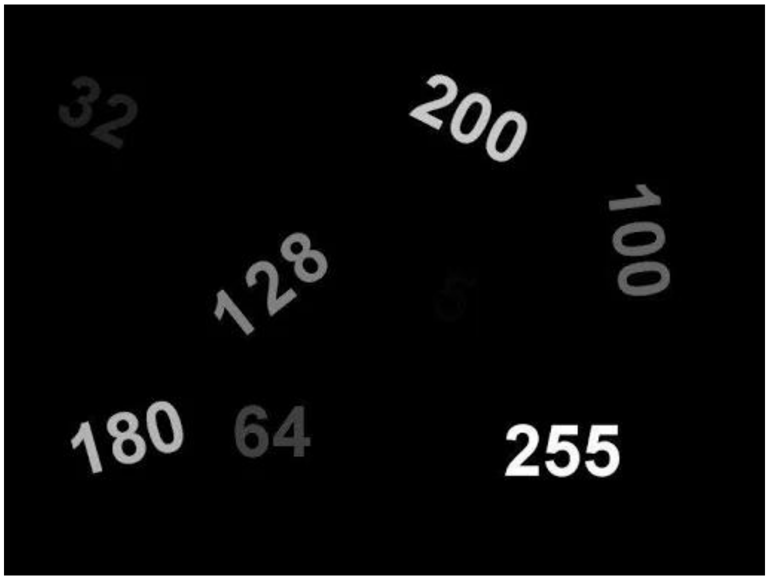
Using thresholding techniques to convert the grayscale image into a binary image. Choose the most good-looking one.



Hint: Try with simple or adaptive or Otsu’s thresholding method.

Ex4.2. Number bounding-box

Given an input image as follows (numbers are pixel intensities). Using morphological operations and contour to draw rectangles surrounding each number (bounding boxes).



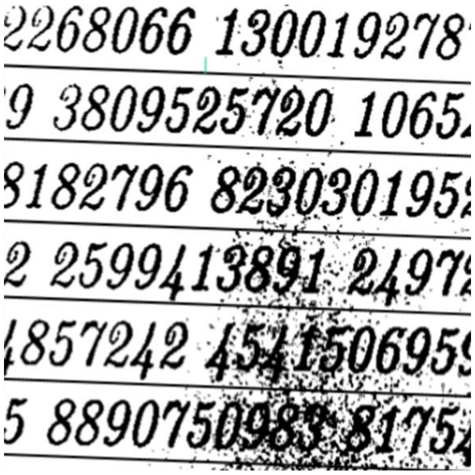
Hint:

(1) Convert to binary image (mask of numbers) using thresholding techniques

(2) Using dilation/closing to connect digits of each numbers (choose a suitable kernel in size)

(3) Find contours and bounding boxes (using methods in Contour features: boundingRect)

Ex4.3. Digit bounding-box
Using morphological operations and contour to draw rectangles surrounding each digit (bounding boxes).



Hint:
(1) Convert to binary image (mask of numbers) using thresholding techniques
(2) Using opening and closing to remove noise and connect broken digits
(3) Find contours and bounding boxes (using methods in Contour features)

HOMEWORK

1. Improve the result of each exercise 4.2-4.3

2. Explore some “Hand detection and finger counting” applications
   https://github.com/kostasthanos/Hand-Detection-and-Finger-Counting