


CSC 3141

IMAGE PROCESSING LABORATORY

08 – Morphological Operations



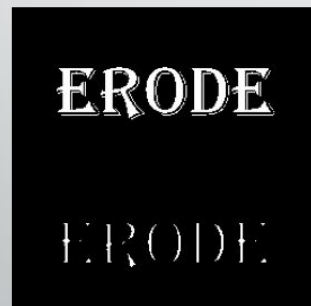
Morphological Transformations

Morphological Transformations

Morphological transformations are some simple operations based on the image shape. It is normally performed on **binary images**. It needs two inputs, the original image and the structuring element or the kernel which decides the nature of operation.

Two basic morphological operators are **Erosion** and **Dilation**. OpenCV contains `cv2.erode()`, `cv2.dilate()`, `cv2.morphologyEx()` as in-built functions to perform the morphological operations easily.

(A) Erosion



(B) Dilation



(C) Opening



(D) Closing



Create custom kernels for morphological operations

```
kernel = cv2.getStructuringElement( cv2_shape, kernel_size )
```

```
# Rectangular Kernel
kernel_1 = np.ones((5,5),np.uint8)

# Rectangular Kernel
kernel_2 = cv2.getStructuringElement(cv2.MORPH_RECT,(5,5))
# Elliptical Kernel
kernel_3 = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(5,5))
# Cross-shaped Kernel
kernel_4 = cv2.getStructuringElement(cv2.MORPH_CROSS,(5,5))
```

5*5 - np.ones

```
[[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]]
```

5*5 - cv2.MORPH_RECT

```
[[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]]
```

5*5 - cv2.MORPH_ELLIPSE

```
[[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]]
```

5*5 - cv2.MORPH_CROSS

```
[[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]]
```


Erosion

The basic idea of erosion is just like soil erosion only, it erodes away the white pixels, typically (boundaries of foreground object). 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).

This is useful for removing small white noises and detaching two connected objects.

```
img = cv2.imread(r'images\j.png', 0)

kernel_1 = np.ones((5, 5), np.uint8)
kernel_2 = cv2.getStructuringElement(cv2.MORPH_RECT, (5, 5))
kernel_3 = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))
kernel_4 = cv2.getStructuringElement(cv2.MORPH_CROSS, (5, 5))

erosion_1 = cv2.erode(img, kernel_1, iterations = 1)
erosion_2 = cv2.erode(img, kernel_2, iterations = 1)
erosion_3 = cv2.erode(img, kernel_3, iterations = 1)
erosion_4 = cv2.erode(img, kernel_4, iterations = 1)
```

Original Image



Erosion 5*5 np.ones



Erosion 5*5 MORPH_RECT



Erosion 5*5 MORPH_ELLIPSE



Erosion 5*5 MORPH_CROSS



Dilation

It is the opposite of erosion. Here, a pixel element is '1' if at least one pixel under the kernel is '1'. Hence 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. This is also useful in joining broken parts of an object.

```
img = cv2.imread(r'images\opening.png', 0)

kernel_1 = np.ones((5, 5), np.uint8)
kernel_2 = cv2.getStructuringElement(cv2.MORPH_RECT, (5, 5))
kernel_3 = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))
kernel_4 = cv2.getStructuringElement(cv2.MORPH_CROSS, (5, 5))

dilation_1 = cv2.dilate(img, kernel_1, iterations = 1)
dilation_2 = cv2.dilate(img, kernel_2, iterations = 1)
dilation_3 = cv2.dilate(img, kernel_3, iterations = 1)
dilation_4 = cv2.dilate(img, kernel_4, iterations = 1)
```

Original Image



Dilation 5*5 np.ones



Dilation 5*5 MORPH_RECT



Dilation 5*5 MORPH_ELLIPSE



Dilation 5*5 MORPH_CROSS



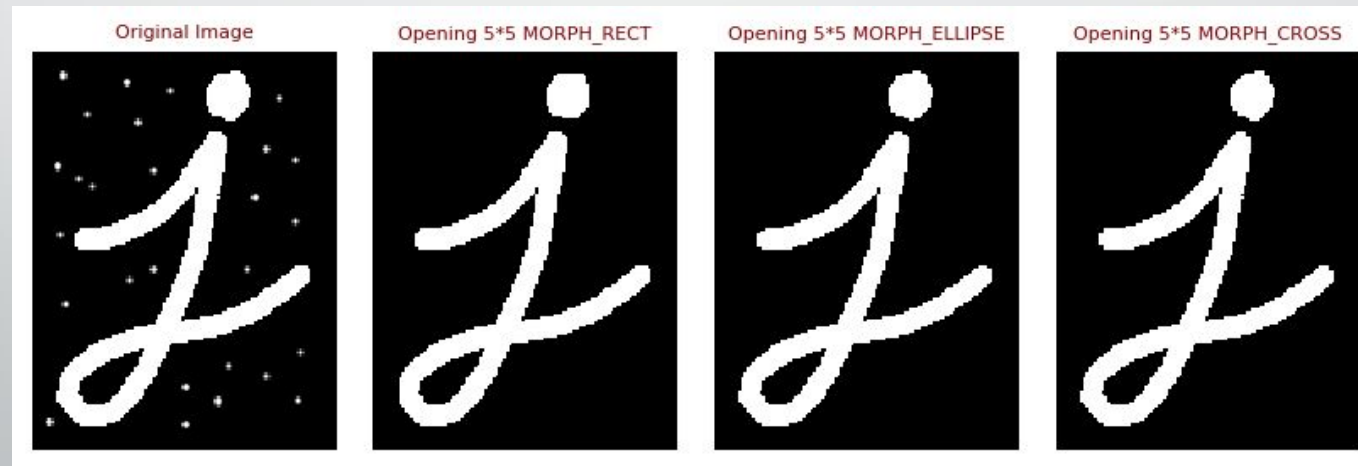
Opening

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

```
img = cv2.imread(r'images\opening.png', 0)

kernel_2 = cv2.getStructuringElement(cv2.MORPH_RECT, (5, 5))
kernel_3 = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))
kernel_4 = cv2.getStructuringElement(cv2.MORPH_CROSS, (5, 5))

opening_2 = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel_2)
opening_3 = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel_3)
opening_4 = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel_4)
```



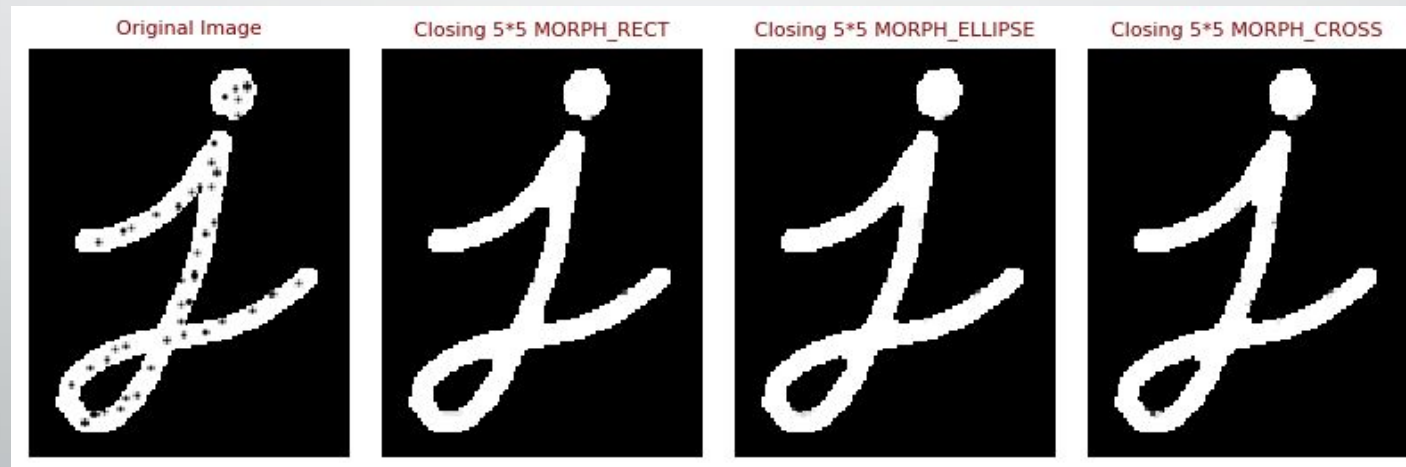
Closing

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.

```
img = cv2.imread(r'images\closing.png', 0)

kernel_2 = cv2.getStructuringElement(cv2.MORPH_RECT, (5, 5))
kernel_3 = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))
kernel_4 = cv2.getStructuringElement(cv2.MORPH_CROSS, (5, 5))

closing_2 = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel_2)
closing_3 = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel_3)
closing_4 = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel_4)
```



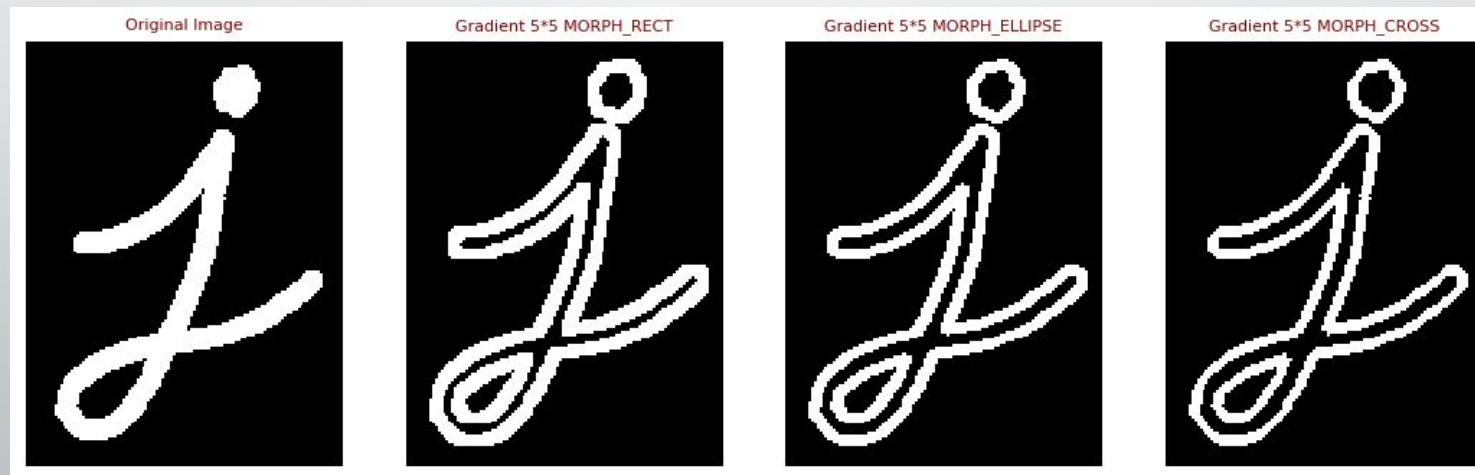
Morphological Gradient

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

```
img = cv2.imread(r'images\j.png', 0)

kernel_2 = cv2.getStructuringElement(cv2.MORPH_RECT, (5, 5))
kernel_3 = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))
kernel_4 = cv2.getStructuringElement(cv2.MORPH_CROSS, (5, 5))

gradient_2 = cv2.morphologyEx(img, cv2.MORPH_GRADIENT, kernel_2)
gradient_3 = cv2.morphologyEx(img, cv2.MORPH_GRADIENT, kernel_3)
gradient_4 = cv2.morphologyEx(img, cv2.MORPH_GRADIENT, kernel_4)
```



Top Hat / White Hat

It is the **difference** between **Opening** of the image and **input** image. This is mainly used for enhancing bright objects of interest in a dark background.

```
img = cv2.imread(r'images\j.png', 0)

kernel_2 = cv2.getStructuringElement(cv2.MORPH_RECT, (9, 9))
kernel_3 = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (9, 9))
kernel_4 = cv2.getStructuringElement(cv2.MORPH_CROSS, (9, 9))

tophat_2 = cv2.morphologyEx(img, cv2.MORPH_TOPHAT, kernel_2)
tophat_3 = cv2.morphologyEx(img, cv2.MORPH_TOPHAT, kernel_3)
tophat_4 = cv2.morphologyEx(img, cv2.MORPH_TOPHAT, kernel_4)
```



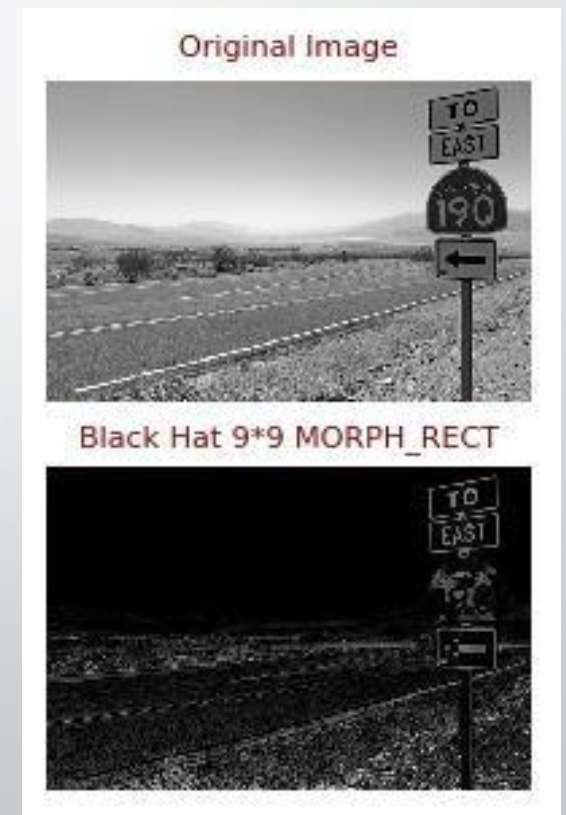
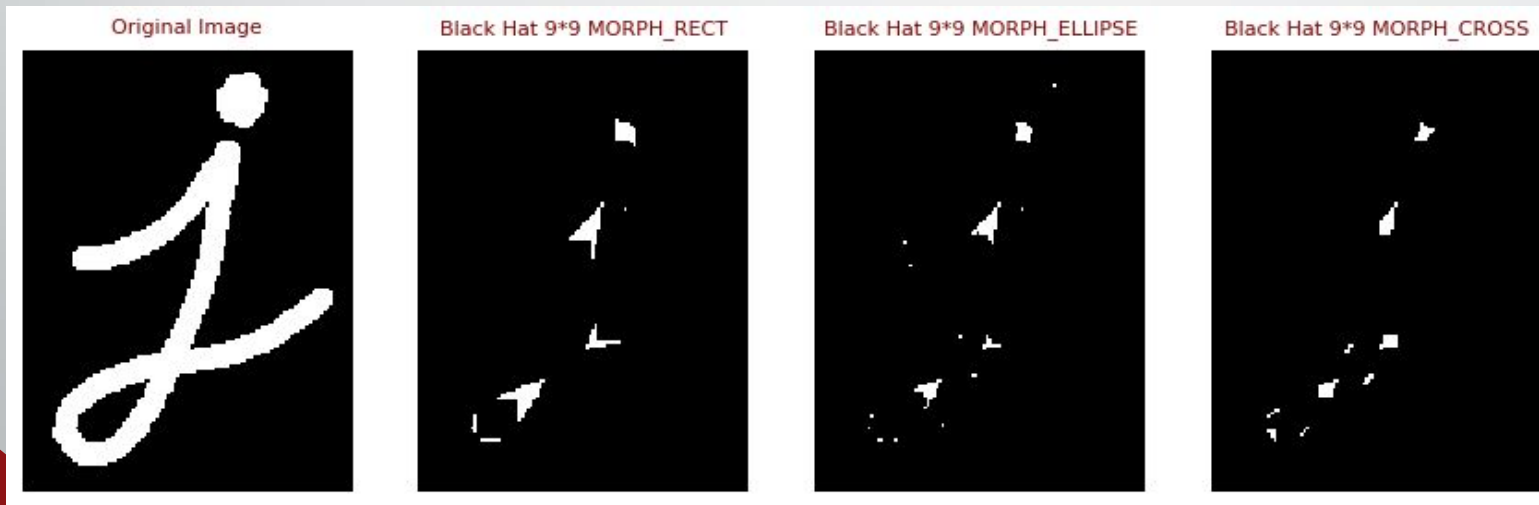
Black Hat

It is the **difference** between the **closing** of the input image and **input** image. This is used for enhancing dark objects of interest in a bright background.

```
img = cv2.imread(r'images\j.png',0)

kernel_2 = cv2.getStructuringElement(cv2.MORPH_RECT, (9,9))
kernel_3 = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (9,9))
kernel_4 = cv2.getStructuringElement(cv2.MORPH_CROSS, (9,9))

blackhat_2 = cv2.morphologyEx(img, cv2.MORPH_BLACKHAT, kernel_2)
blackhat_3 = cv2.morphologyEx(img, cv2.MORPH_BLACKHAT, kernel_3)
blackhat_4 = cv2.morphologyEx(img, cv2.MORPH_BLACKHAT, kernel_4)
```





Contours

Contours

Contours can be explained as a curve joining all the continuous points (along the boundary), having same color or intensity. The contours are a useful tool for shape analysis and object detection and recognition.

```
image, contours, hierarchy = cv2.findContours(thresh,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
```

Important

1. For better accuracy, use **binary images**. So before finding contours, apply threshold or canny edge detection.
2. `findContours()` function modifies the source image. So if you want source image even after finding contours, already store it to some other variables.
3. In OpenCV, finding contours is like finding white object from black background. So remember, object to be found should be white and background should be black.

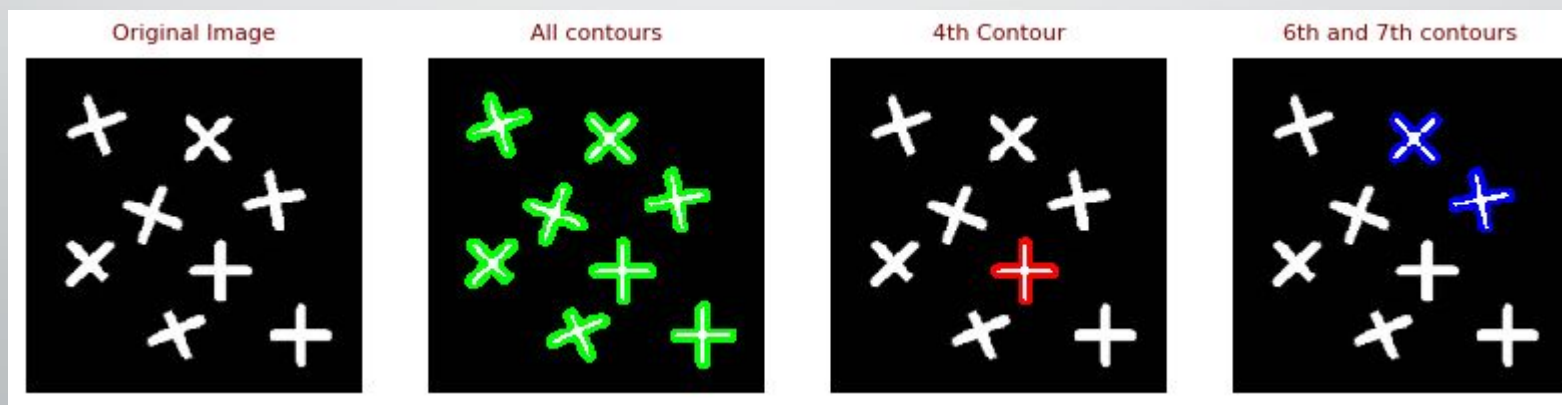
Drawing Contours

```
img = cv2.imread(r'images\plus.jpg',1)
img__ = img.copy()
img___ = img.copy()
img_org = img.copy()

img_g = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

ret,thresh = cv2.threshold(img_g,50,255,0)
image, contours, hierarchy = cv2.findContours(thresh,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)

#img1 = cv2.drawContours(img, contours, index, border_color, line_width)
img1 = cv2.drawContours(img, contours, -1, (0,255,0), 2)
img2 = cv2.drawContours(img__, contours, 3, (255,0,0), 2)
cnt = contours[5]
cnt1 = contours[6]
#img3 = cv2.drawContours(img___, [cnt], 0, (0,0,255), 2)
img3 = cv2.drawContours(img___, [cnt,cnt1], -1, (0,0,255), 2)
```



Contour Features

Area

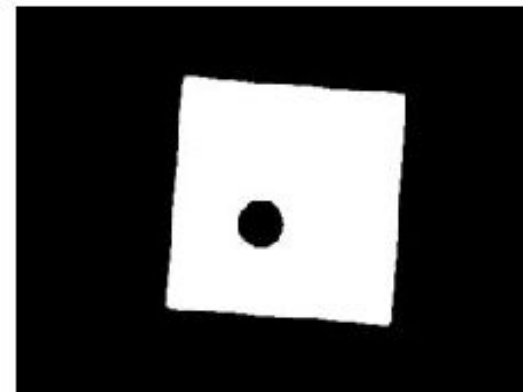
```
area = cv2.contourArea(contours[0])

#writing text
position = (20,20) #y,x
text = "Sqaure Area : "+str(area) +" pixel^2"
cv2.putText(
    img9, #numpy array on which text is written
    text, #text
    position, #position at which writing has to start
    cv2.FONT_HERSHEY_SIMPLEX, #font family
    0.5, #font size
    (255, 255, 0, 255), #font color
    2) #font stroke
```

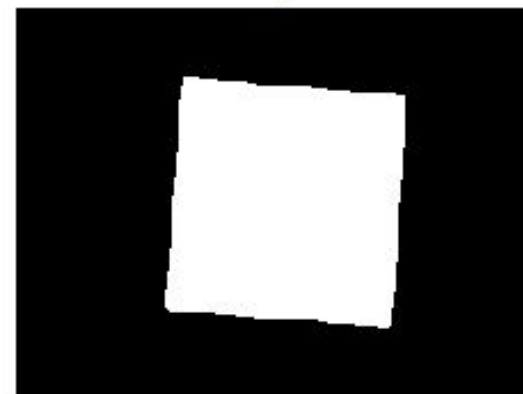
Perimeter

```
#Second argument - whether a closed contour or just a curve.
perimeter = cv2.arcLength(contours[0],True)
perimeter = "{0:.2f}".format(perimeter)
position1 = (20,195) #x,y
text1 = "Sqaure perimeter : "+str(perimeter) +" pixels"
#x,y
plt.text(20, 195, text1, fontsize = 8,color = 'r')
plt.imshow(img9,'gray')
```

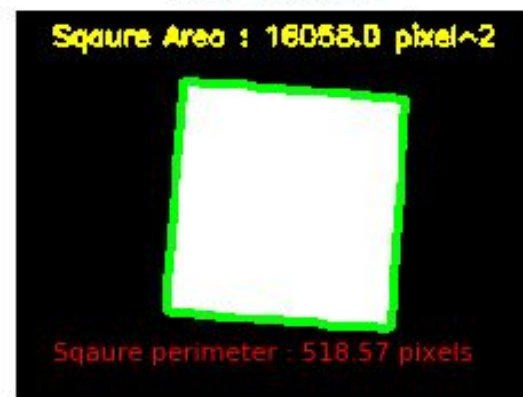
Original Image



closing with 21*21
MORPH_RECT



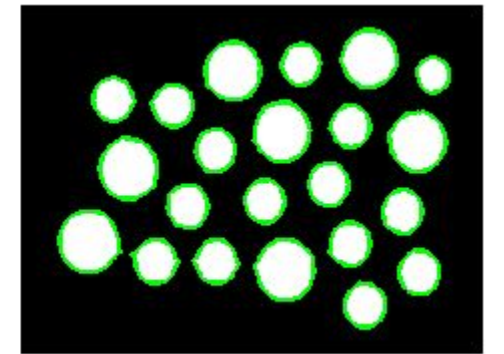
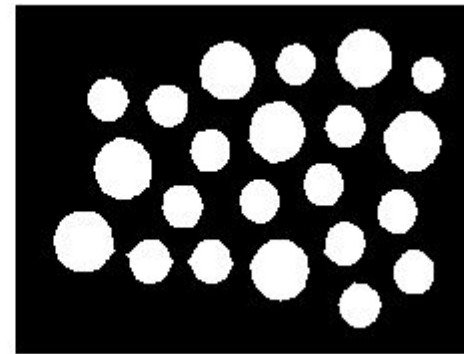
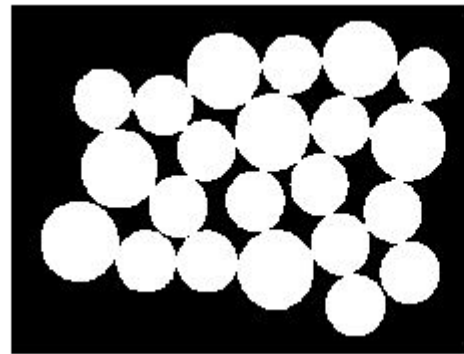
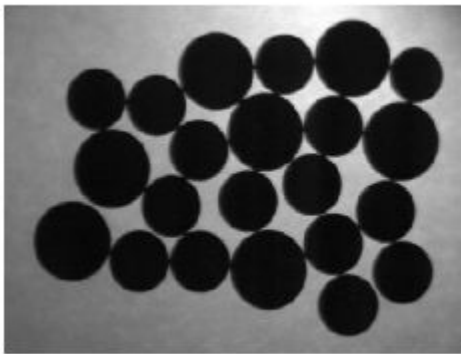
Area Calculated



Assignment - Count the Coins

Steps

- # define SE
- # perform Thresholding
- # perform morphological transformation
- # count the coins
- # calculate the total area covered by coins





Practical Use Cases - Contours

Element Counting

Area approximation

Detect the borders of objects

Objects localization

Object perimeter calculation



Canny Edge Detection

Canny Edge Detection

First argument is our input image. Second and third arguments are our minVal and maxVal respectively. Fourth argument is aperture_size. It is the size of Sobel kernel used for find image gradients. By default it is 3.

```
img = cv2.imread(r'images\double_edge.jpg',0)

#edges_img = cv2.Canny(img, minThreshold, maxThreshold)
edges = cv2.Canny(img,100,200)
```



Ref: https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html

Ref: <https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123>



Additional

Sobel Operators

In the previous lecture note (L07), Sobel output data type was `cv2.CV_8U` or `np.uint8`. But there is a slight problem with that. Black-to-White transition is taken as Positive slope (it has a positive value) while White-to-Black transition is taken as a Negative slope (It has negative value). So when you convert data to `np.uint8`, all negative slopes are made zero. In simple words, you miss that edge.

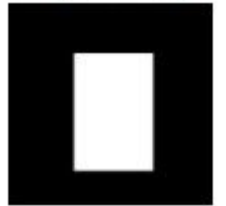
If you want to detect both edges, better option is to keep the output data type to some higher forms, like `cv2.CV_16S`, `cv2.CV_64F` etc, take its absolute value and then convert back to `cv2.CV_8U`.

```
img = cv2.imread(r'images\double_edge.jpg',0)

# Output dtype = cv2.CV_8U
sobelx8u = cv2.Sobel(img,cv2.CV_8U,1,0,ksize=3)

# Output dtype = cv2.CV_64F. Then take its absolute and convert to cv2.CV_8U
sobelx64f = cv2.Sobel(img,cv2.CV_64F,1,0,ksize=3)
abs_sobel64f = cv2.convertScaleAbs(sobelx64f)
sobel_8u = np.uint8(abs_sobel64f)
```

Original Image



cv2.CV_8U



cv2.CV_64F



abs(cv2.CV_64F)



np.uint8(abs(cv2.CV_64F))



— END —

