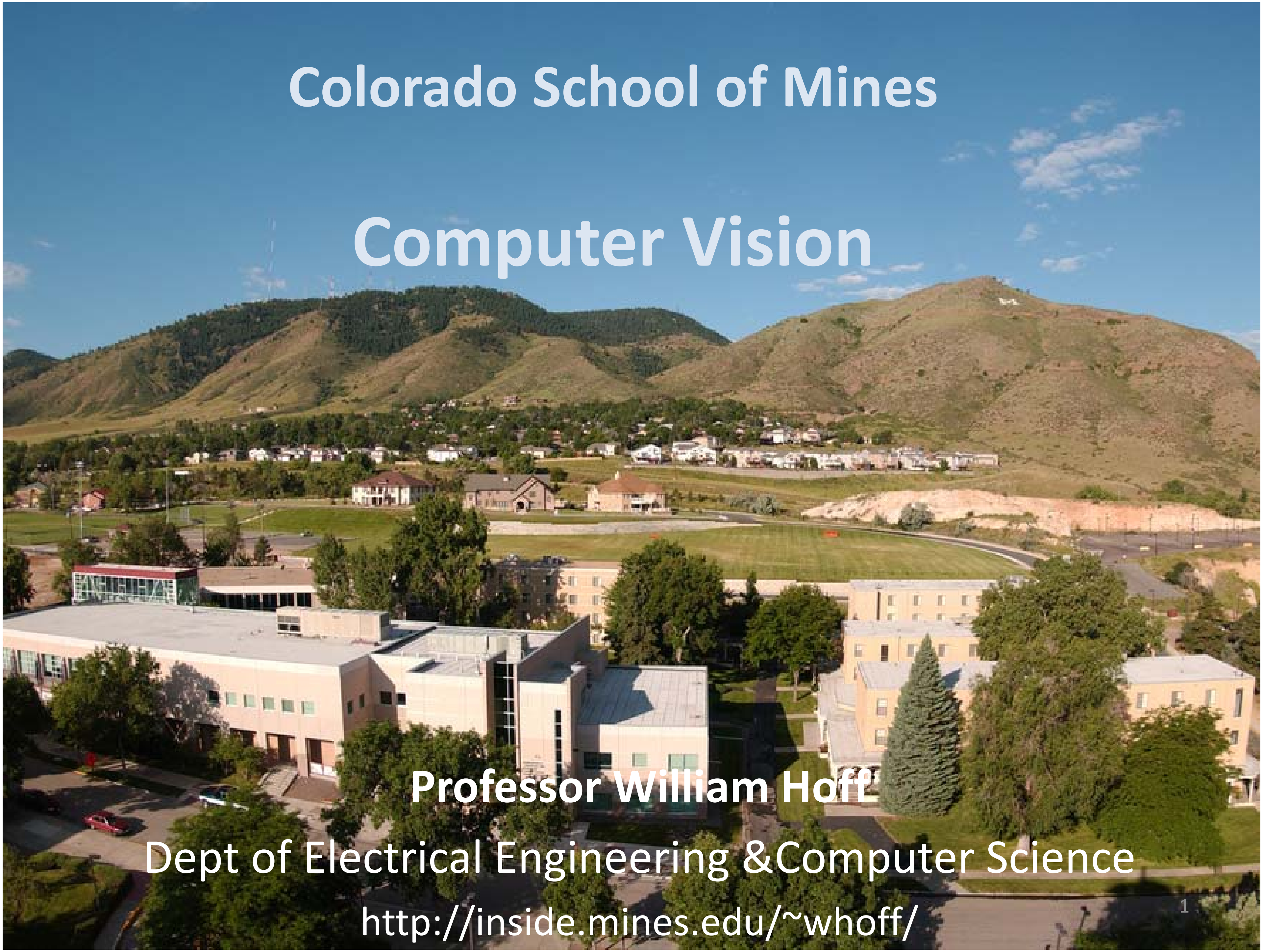


Colorado School of Mines

Computer Vision

Professor William Hoff

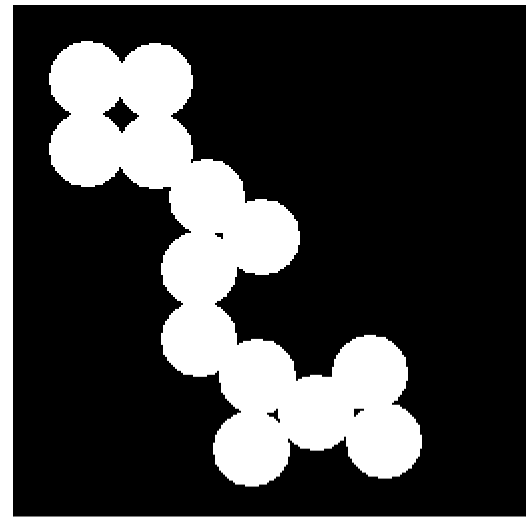
Dept of Electrical Engineering &Computer Science

http://inside.mines.edu/~whoff/

Binary Image Processing

Binary Images

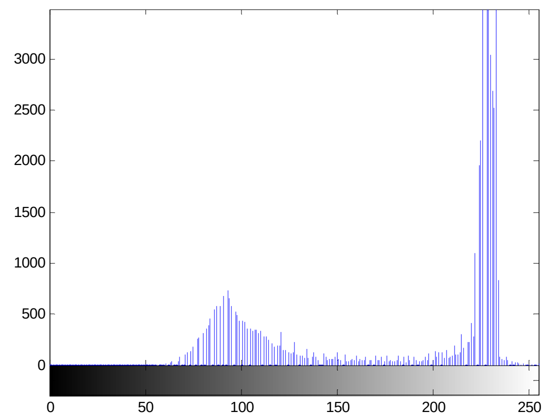
- "Binary" means
  - 0 or 1 values only
  - Or, true/false
- Obtained from
  - Thresholding gray level images
  - Or, the result of feature detectors
- Often want to count or measure shape of 2D binary image regions



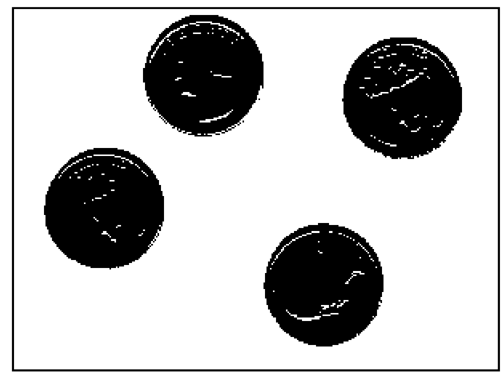
Thresholding

- Convert gray scale image to binary (0s and 1s)
- Bright pixels are mapped to 1s, dark pixels are mapped to 0s
- Need to pick a threshold value

https://www.bogotobogo.com/Matlab/images/MATLAB_DEMO_IMAGES/eight.tif



Pick, say 200 for the threshold



```
B = img > 200   # Creates a boolean image

from matplotlib import pyplot as plt
plt.figure()    # Matplotlib can display boolean images
plt.imshow(B, cmap="gray")
plt.show()
```

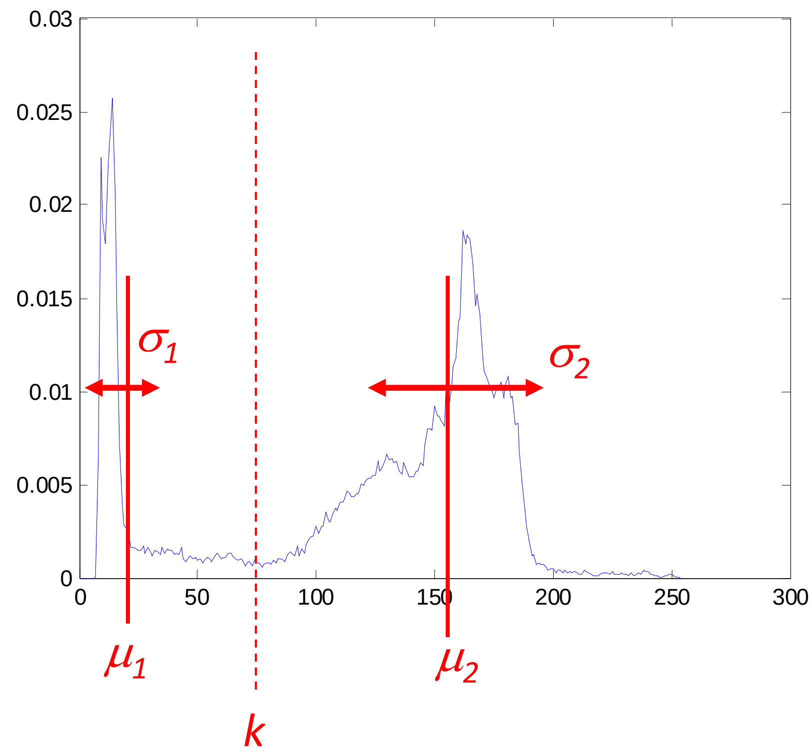
Otsu's Method for Global Thresholding

- Find a threshold that minimizes the variance within groups

$$\sigma_W^2 = P_1\, \sigma_1^2 + P_2\, \sigma_2^2$$

- where

$$P_1 = \sum_{i=0}^{k} p_i, \quad P_2 = \sum_{i=k+1}^{L-1} p_i$$



  - Intuitively, we want to have each group to be tightly clustered

OpenCV Thresholding Functions

```
# Note - if you have a color image, you need to convert it to grayscale.
gray_img = cv2.cvtColor(bgr_img, cv2.COLOR_BGR2GRAY)
```

```
# Do thresholding using a preset threshold.  Returns image of type 8-bit unsigned.
_, binary_img = cv2.threshold(gray_img, thresh=127, maxval=255, type=cv2.THRESH_BINARY)
```

Use underscore to ignore the return argument (which is just the threshold we provided)

```
# Do thresholding using Otsu's algorithm.  The computed threshold value is returned.
thresh, binary_img = cv2.threshold(
    gray_img, thresh=0, maxval=255, type=cv2.THRESH_BINARY + cv2.THRESH_OTSU)
```

Input threshold is ignored

Note that we can combine flags (see next slide)

OpenCV Thresholding Flags

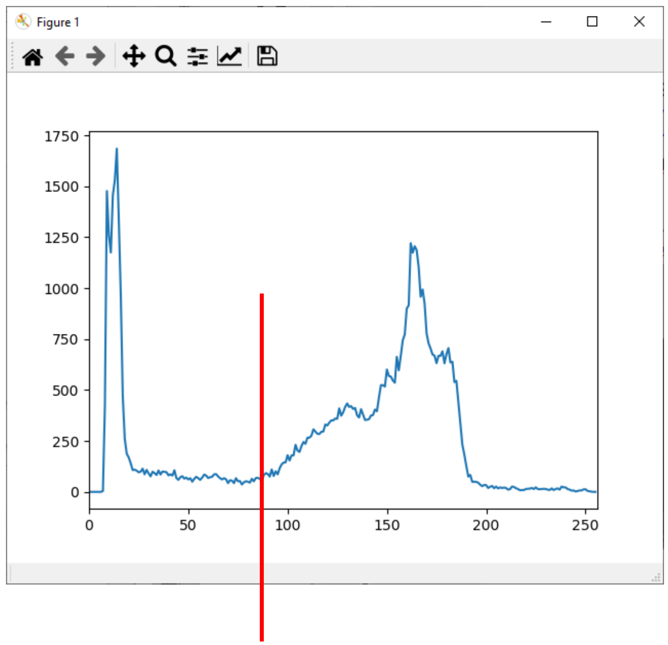
Usual type, where lighter values are mapped to white	THRESH_BINARY Python: cv.THRESH_BINARY	$\texttt{dst}(x,y) = \begin{cases} \texttt{maxval} & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0 & \text{otherwise} \end{cases}$
If you need to map darker values to white	THRESH_BINARY_INV Python: cv.THRESH_BINARY_INV	$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{maxval} & \text{otherwise} \end{cases}$
	THRESH_TRUNC Python: cv.THRESH_TRUNC	$\texttt{dst}(x,y) = \begin{cases} \texttt{threshold} & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$
	THRESH_TOZERO Python: cv.THRESH_TOZERO	$\texttt{dst}(x,y) = \begin{cases} \texttt{src}(x,y) & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0 & \text{otherwise} \end{cases}$
	THRESH_TOZERO_INV Python: cv.THRESH_TOZERO_INV	$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$
	THRESH_MASK Python: cv.THRESH_MASK	
Compute threshold automatically	THRESH_OTSU Python: cv.THRESH_OTSU	flag, use Otsu algorithm to choose the optimal threshold value
	THRESH_TRIANGLE Python: cv.THRESH_TRIANGLE	flag, use Triangle algorithm to choose the optimal threshold value

From documentation for “threshold” at https://docs.opencv.org

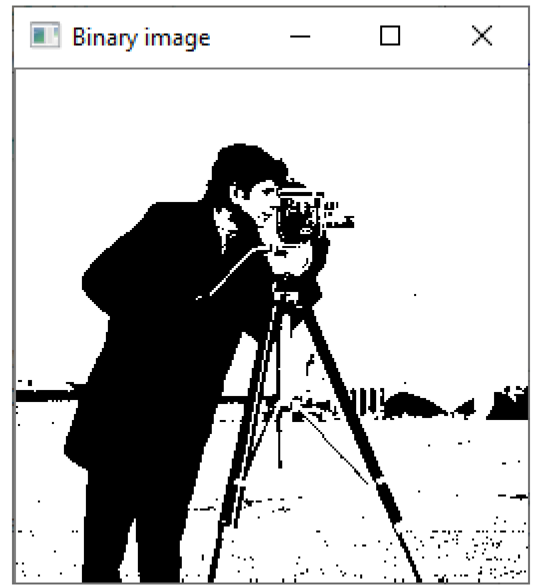
Otsu Thresholding Example

- Image is from https://www.bogotobogo.com/Matlab/images/MATLAB_DEMO_IMAGES

cameraman.tif



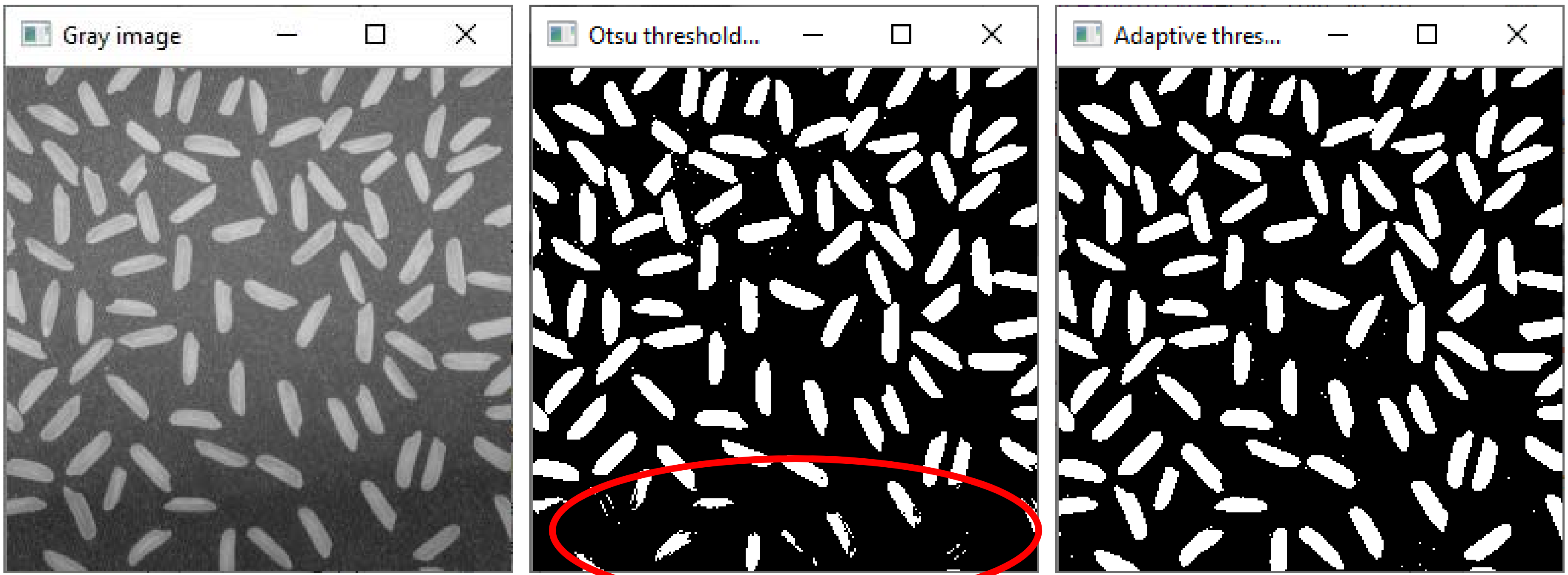
Computed threshold = 88.0



Adaptive Thresholding

- Do a threshold over small windows
- Within each window, compare each pixel to a local mean of the window
- Useful when lighting is uneven

```
binary_img = cv2.adaptiveThreshold(
    src=gray_img,
    maxValue=255,  # output value where condition met
    adaptiveMethod=cv2.ADAPTIVE_THRESH_MEAN_C,
    thresholdType=cv2.THRESH_BINARY,  # threshold_type
    blockSize=51,  # neighborhood size (a large odd number)
    C=-10)  # a constant to subtract from mean
```

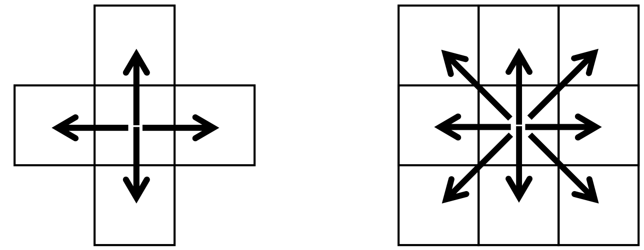


https://www.bogotobogo.com/Matlab/images/MATLAB_DEMO_IMAGES/rice.png

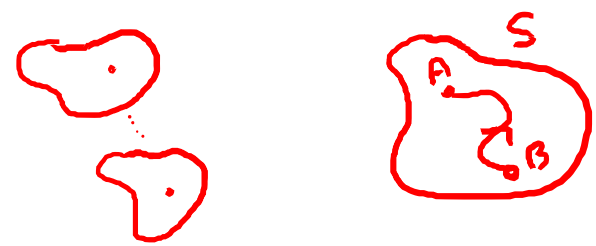
Connected Components

- Define adjacency
  - 4-adjacent
  - 8-adjacent

4-connected 8-connected

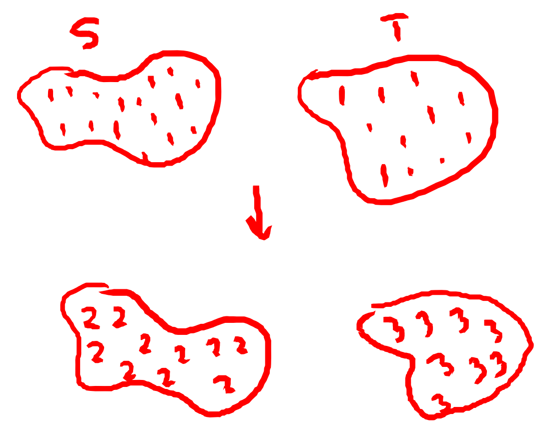


- Two pixels are connected in S if there is a path between them consisting entirely of pixels in S



- S is a (4- or 8-) connected component ("blob") if there exists a path between every pair of pixels

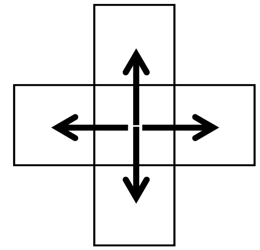
- "Labeling" is the process of assigning the same label number to every pixel in a connected component



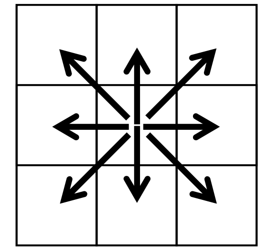
Example

- Hand label simple binary image

4-connected



8-connected



	1	1				
	1	1				
			1	1		
	1		1	1		
	1		1			1

Binary image

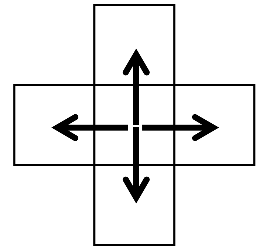
Labeled image (4-connected)

Labeled image (8-connected)

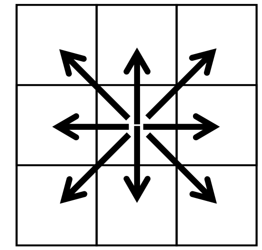
Example

- Hand label simple binary image

4-connected



8-connected



	1	1				
	1	1				
			1	1		
	1		1	1		
	1		1			1

Binary image

	1	1				
	1	1				
			2	2		
	3		2	2		
	3		2			4

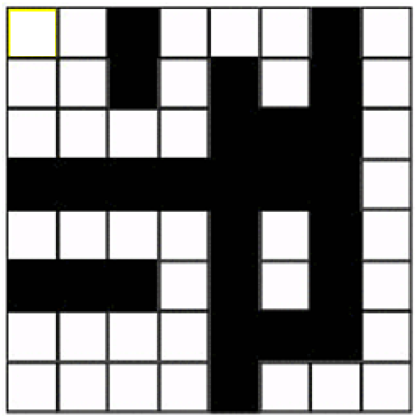
Labeled image (4-connected)

	1	1				
	1	1				
			1	1		
	2		1	1		
	2		1			3

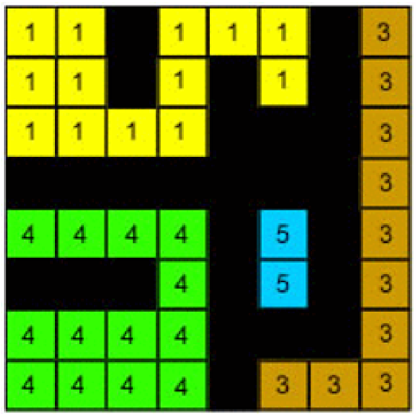
Labeled image (8-connected)

Another example

Binary image



Final labels



- Connected component labeling is fast
- It can be done with only two passes through the image
  - For a nice animation of the two pass algorithm, see https://iq.opengenus.org/connected-component-labeling/

OpenCV Connected Component Labeling

- Find connected components in a binary image

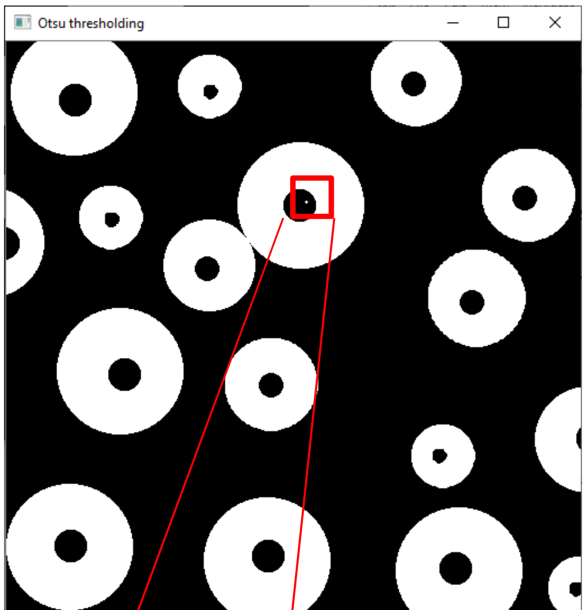
```
# Find connected component labels for all white blobs.
num_white_labels, labels_white_img = cv2.connectedComponents(binary_img)
print("Number of white labels = ", num_white_labels)

# Scale (for display purposes only).
# Min value is set to alpha, max value to beta.
labels_display = cv2.normalize(
    src = labels_white_img, dst = None, alpha = 0, beta = 255,
    norm_type = cv2.NORM_MINMAX, dtype = cv2.CV_8U)

cv2.imshow("Labels image", labels_display)
```

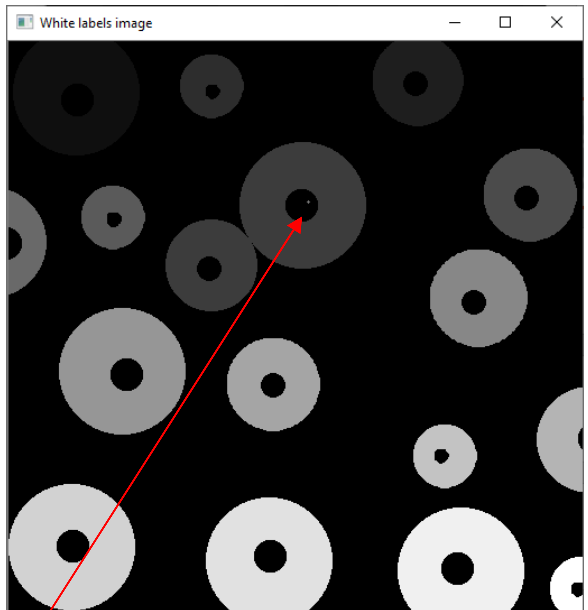
- If you need to find connected components for *black* blobs, complement the binary image

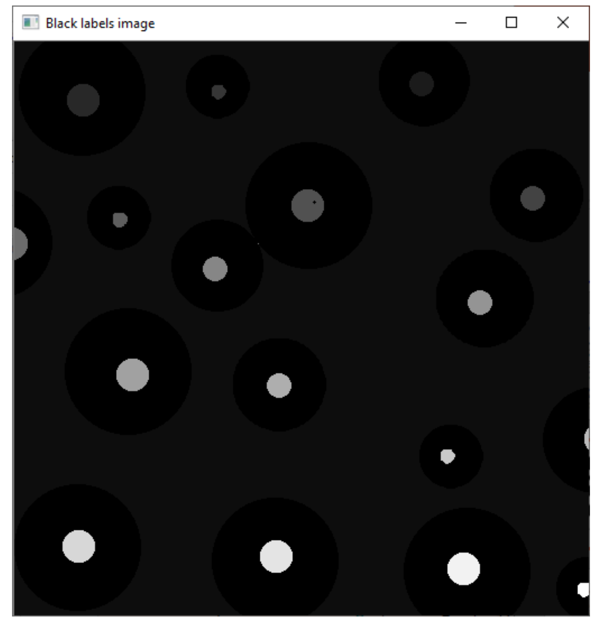
```
# Find connected component labels for all black blobs.
num_black_labels, labels_black_img = cv2.connectedComponents(cv2.bitwise_not(binary_img))
```

Example

- Labels are displayed as gray levels, 1..N



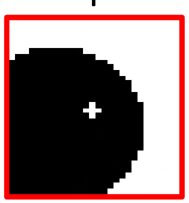




Input binary image

White labels

Black labels



Note the little white blob!

Number of white labels = 18

Number of black labels = 20

Binary Image Morphology

- We can "clean up" a binary image, before finding connected components
  - Get rid of tiny regions or holes
- We'll look at "morphological operations":
  - Dilation and erosion
  - Opening and closing
- Operations are performed with a "structuring element" S
  - S is a small binary image
  - Like a filter mask

Dilation

1	1	1
1	**1**	1
1	1	1

S

- Defined as

$$B \oplus S = \bigcup_{b \in B} S_b$$

- where
  - S_b is the structuring element S, shifted to b
- Procedure
  - Sweep S over B
  - Everywhere the origin of S touches a 1, OR S with the result image
- Expands regions

		1		1	
		1	1		

B

$B \oplus S$

Dilation

1	1	1
1	**1**	1
1	1	1

S

- Defined as

$$B \oplus S = \bigcup_{b \in B} S_b$$

- where
  - S_b is the structuring element S, shifted to b
- Procedure
  - Sweep S over B
  - Everywhere the origin of S touches a 1, OR S with the result image
- Expands regions

		1		1	
		1	1		

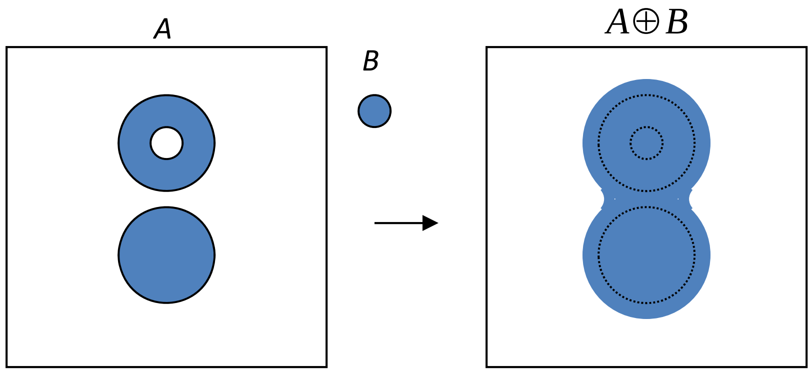
B

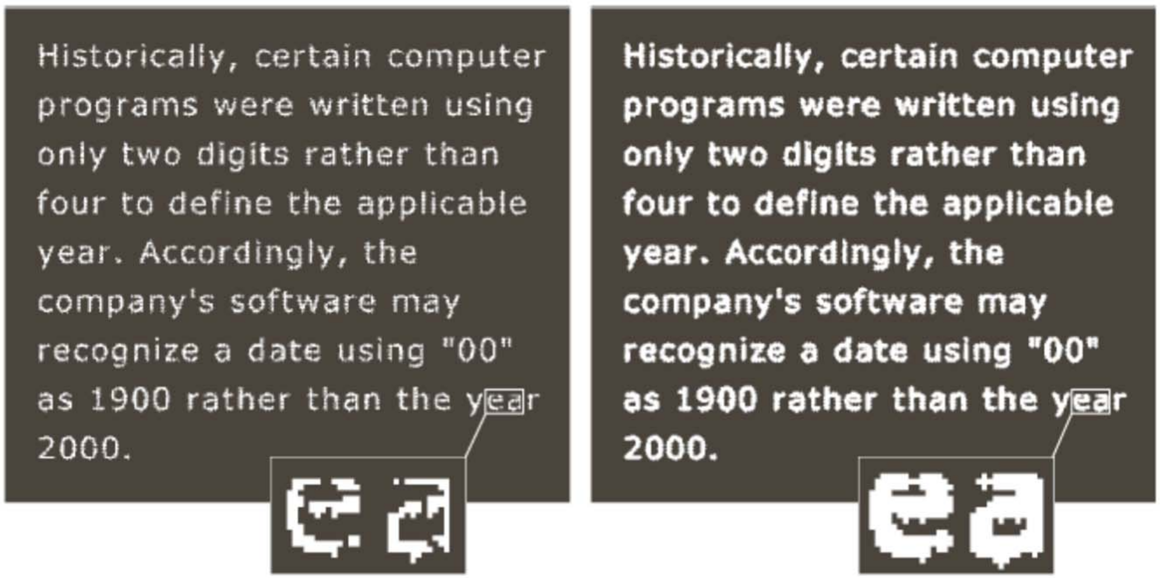
	1	1	1	1	1
	1	1	1	1	1
	1	1	1	1	1
	1	1	1	1	

$B \oplus S$

Dilation

- Dilation fills in holes, thickens thin parts, grows object




Historically, certain computer programs were written using only two digits rather than four to define the applicable year. Accordingly, the company's software may recognize a date using "00" as 1900 rather than the year 2000.

Historically, certain computer programs were written using only two digits rather than four to define the applicable year. Accordingly, the company's software may recognize a date using "00" as 1900 rather than the year 2000.

0	1	0
1	1	1
0	1	0

a c
b

FIGURE 9.7
(a) Sample text of poor resolution with broken characters (see magnified view).
(b) Structuring element.
(c) Dilation of (a) by (b). Broken segments were joined.

Erosion

- Defined as

$$B \Theta S = \{ b \mid b + s \in B, \forall s \in S \}$$

- Procedure
  - Sweep S over B
  - Everywhere S is completely contained in B, output a 1 at the origin of S
- Shrinks regions

S

1	1	1
1	**1**	1
1	1	1

B

	1	1	1	1	1
	1	1	1	1	1
	1	1	1	1	1
	1	1	1	1	

$B \Theta S$

Erosion

- Defined as

$$B \Theta S = \{ b \mid b + s \in B, \forall s \in S \}$$

- Procedure
  - Sweep S over B
  - Everywhere S is completely contained in B, output a 1 at the origin of S
- Shrinks regions

S

1	1	1
1	**1**	1
1	1	1

B

	1	1	1	1	1
	1	1	1	1	1
	1	1	1	1	1
	1	1	1	1	

$B \Theta S$

		1	1	1	
		1	1		

Openings and Closings

- Opening
  - Erosion followed by dilation
  - Eliminate small regions and projections

$$B \circ S = \left(B \ominus S \right) \oplus S$$

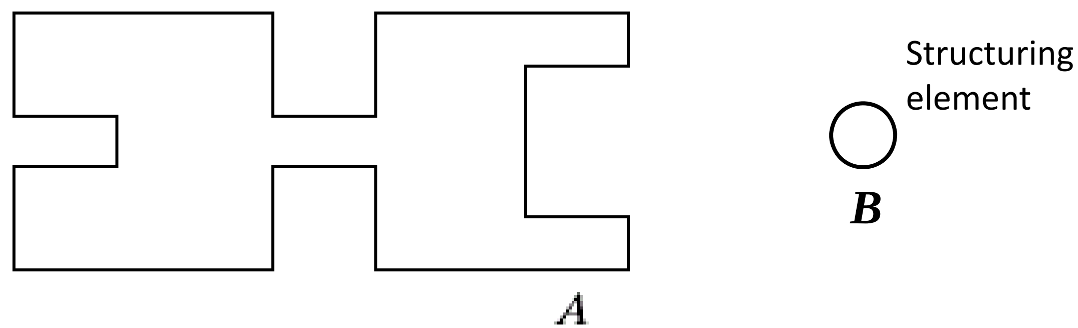
Advantage of openings and closings: doesn't change size of large regions

- Closing
  - Dilation followed by erosion
  - Fill in small holes and gaps

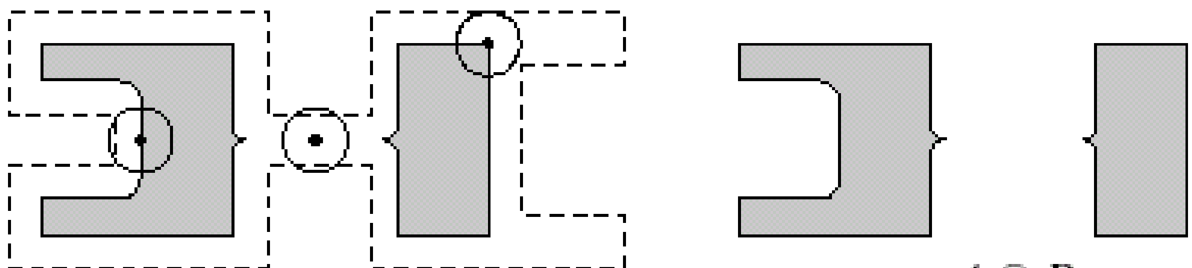
$$B \bullet S = \left(B \oplus S \right) \ominus S$$

Example - opening

Eliminate small regions and projections

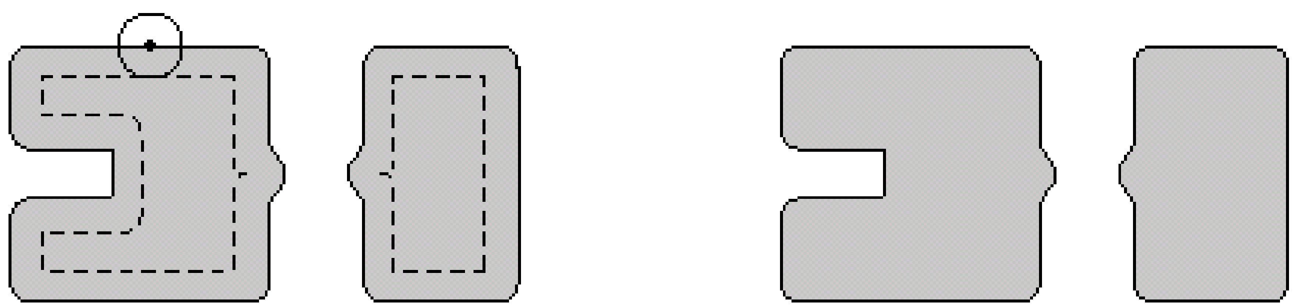


erosion



$A \ominus B$

dilation



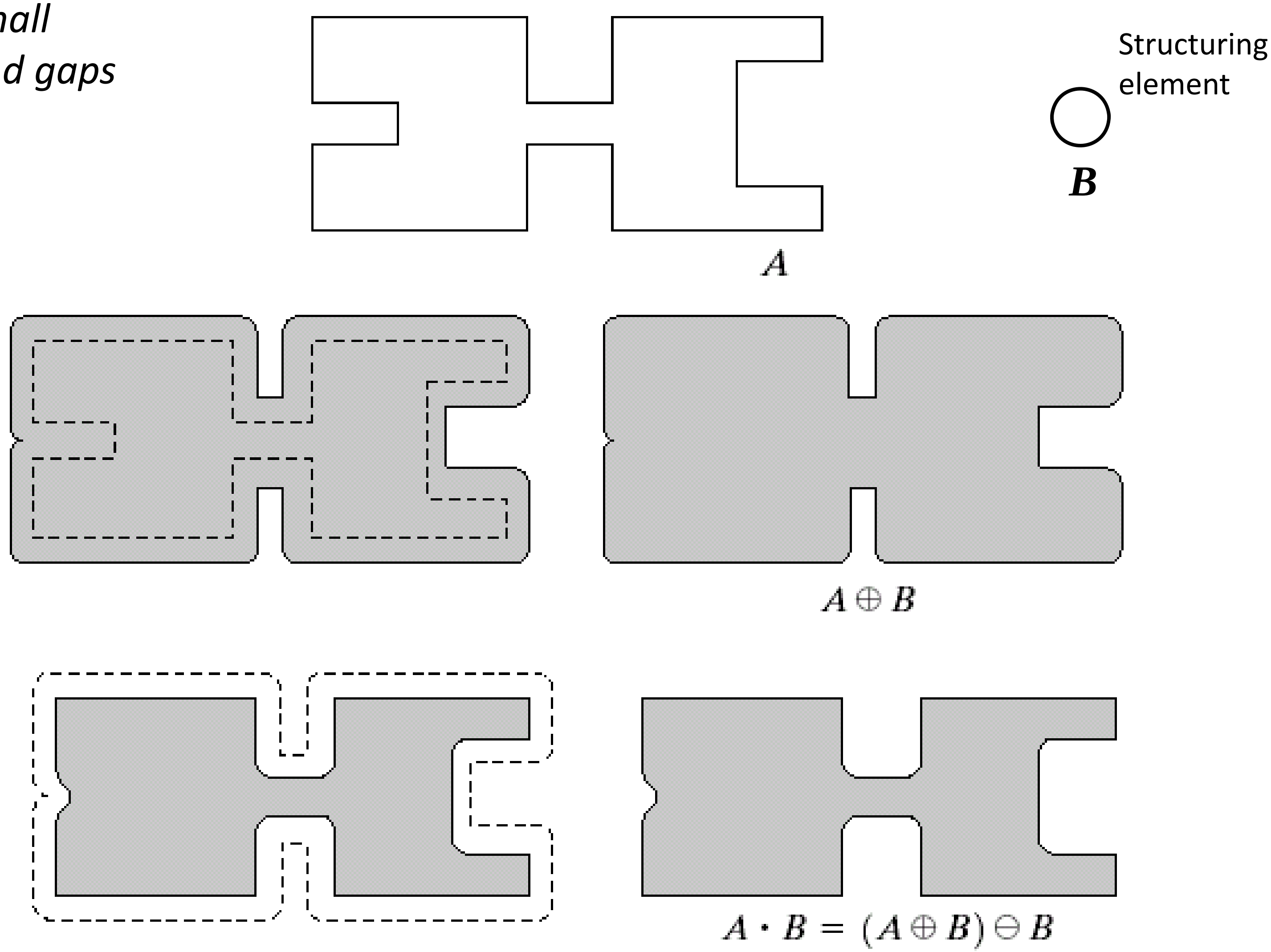
$A \circ B = (A \ominus B) \oplus B$

Example - closing

Fill in small holes and gaps

dilation

erosion



OpenCV

- To create a structuring element, or “kernel”

```
# Create a 5x5 square box filter.
kernel = np.ones((5, 5), np.uint8)
print(kernel)

# Or, make a disk, if we want something more well rounded.
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))
```

- Opening

```
filtered_img = cv2.morphologyEx(binary_img, cv2.MORPH_OPEN, kernel)
```

- Closing

```
filtered_img = cv2.morphologyEx(binary_img, cv2.MORPH_CLOSE, kernel)
```

Example – real image

- Segment all the dark regions in the lower half of the image "bag.png"
  - Namely, generate a binary (or "logical") image which is white (1) in the regions of interest, and black (0) elsewhere
  - Want:
    - No gaps in the regions
    - No extraneous white pixels in the background
- Then do connected component labeling on these regions

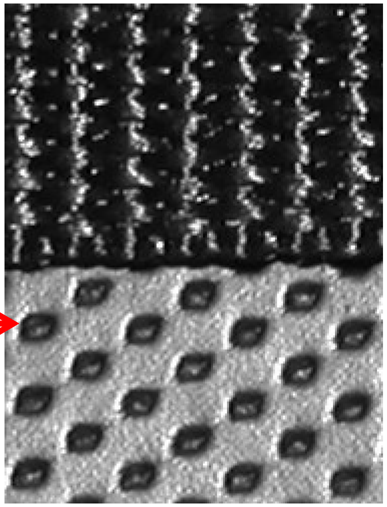


Image "bag.png" from https://www.bogotobogo.com/Matlab/images/MATLAB_DEMO_IMAGES/

For this example, ignore the upper half of the image

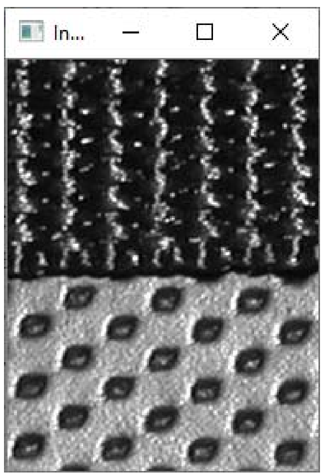
Approach

```python
# Complement the binary image so that the black regions are now white.
binary_img = cv2.bitwise_not(binary_img)

# Clean up using opening + closing.
ksize = 5
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (ksize, ksize))
binary_img = cv2.morphologyEx(binary_img, cv2.MORPH_OPEN, kernel)
binary_img = cv2.morphologyEx(binary_img, cv2.MORPH_CLOSE, kernel)
cv2.imshow("Filtered image", binary_img)

# Find connected component labels for all white blobs.
num_white_labels, labels_white_img = cv2.connectedComponents(binary_img)
```

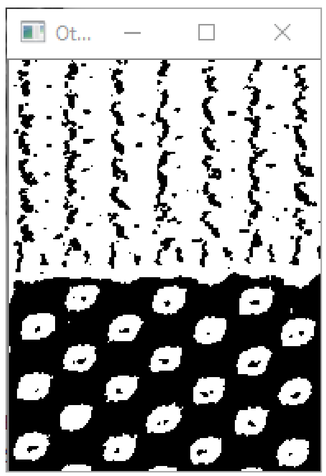
You can find the value of "ksize" experimentally

You can use a different structuring element for opening and closing (not typical)

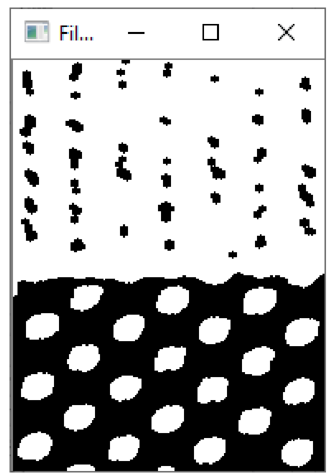
Number of white labels = 27



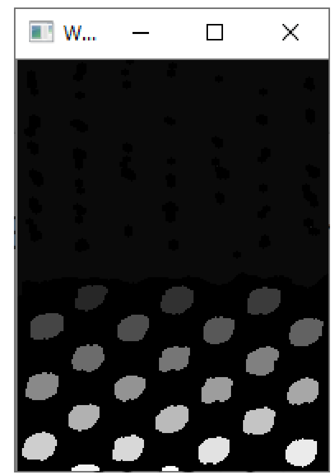
Original "bag.png" image



Binary image after Otsu thresholding



After morphological operations



After connected component labeling. Each gray level indicates a different label of a foreground object

Region Properties

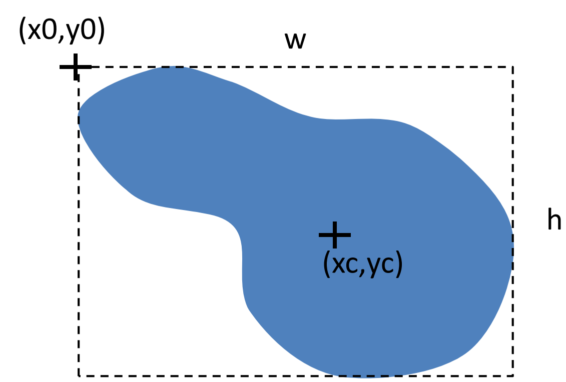
- Area

$$A = \sum_{(r,c)\in R} 1$$

- Centroid

$$\bar{r} = \frac{1}{A} \sum_{(r,c)\in R} r, \quad \bar{c} = \frac{1}{A} \sum_{(r,c)\in R} c$$

- Bounding box
  - The smallest rectangle containing the region
  - Can be specified by
    - The location of the upper left corner
    - The width and height



OpenCV - computing region properties

- Get connected components and region properties

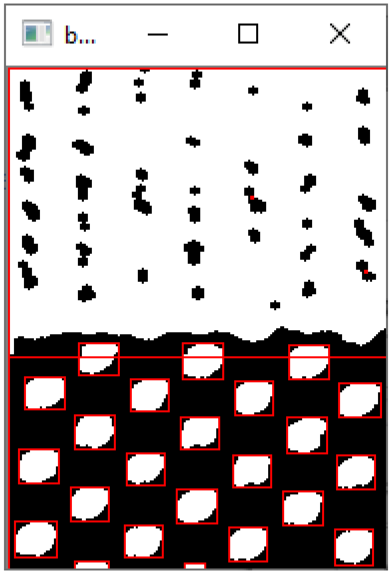
```
num_labels, labels_img, stats, centroids = cv2.connectedComponentsWithStats(binary_img)
```

- Print centroids and areas

```
for n in range(num_labels):
    xc, yc = centroids[n]
    area = stats[n, cv2.CC_STAT_AREA]
    print("Region %d, centroid: (%f,%f), area = %d" % (n, xc, yc, area))
```

- Display bounding boxes

```
bgr_image_display = cv2.cvtColor(binary_img, cv2.COLOR_GRAY2BGR)
for stat, centroid in zip(stats, centroids):
    x0 = stat[cv2.CC_STAT_LEFT]
    y0 = stat[cv2.CC_STAT_TOP]
    w = stat[cv2.CC_STAT_WIDTH]
    h = stat[cv2.CC_STAT_HEIGHT]
    bgr_image_display = cv2.rectangle(
        img=bgr_image_display, pt1=(x0, y0), pt2=(x0 + w, y0 + h),
        color=(0, 0, 255), thickness=1)
cv2.imshow("boxes", bgr_image_display)
```

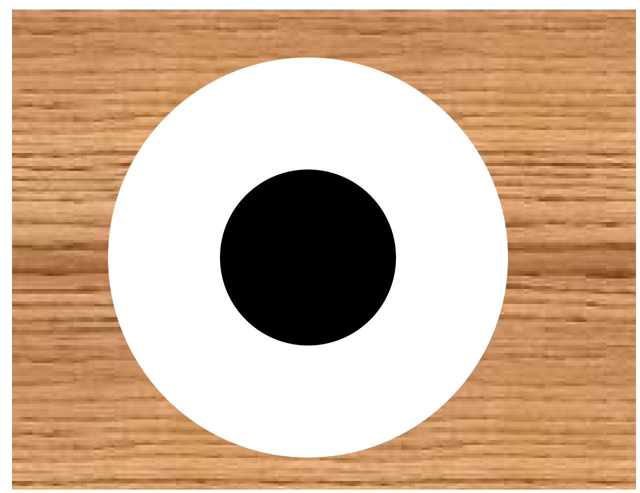


Concentric contrasting circle (CCC) target

- The target is a white ring surrounding a black dot

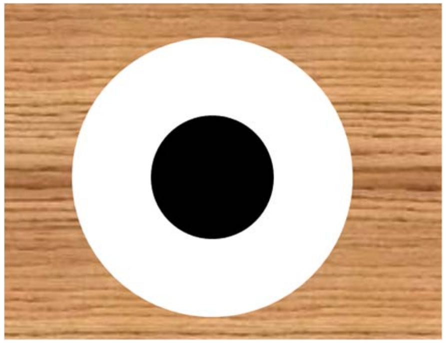
- This feature is fairly unique in the image, because the centroid of the white ring will coincide with the centroid of the black dot

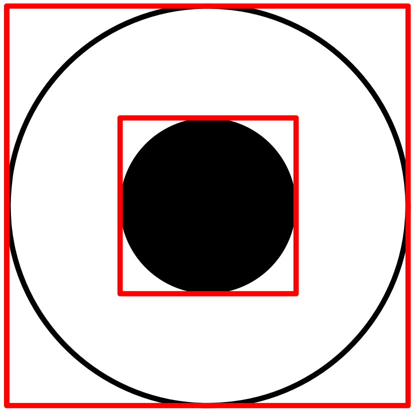
- You can automatically find the target by finding a white region whose centroid coincides with the centroid of a black region



CCC targets (continued)

- For more discrimination power, you can also place constraints on the binary regions, e.g.,
  - The white area must be greater than the black area)
  - The white bounding box must enclose the black bounding box





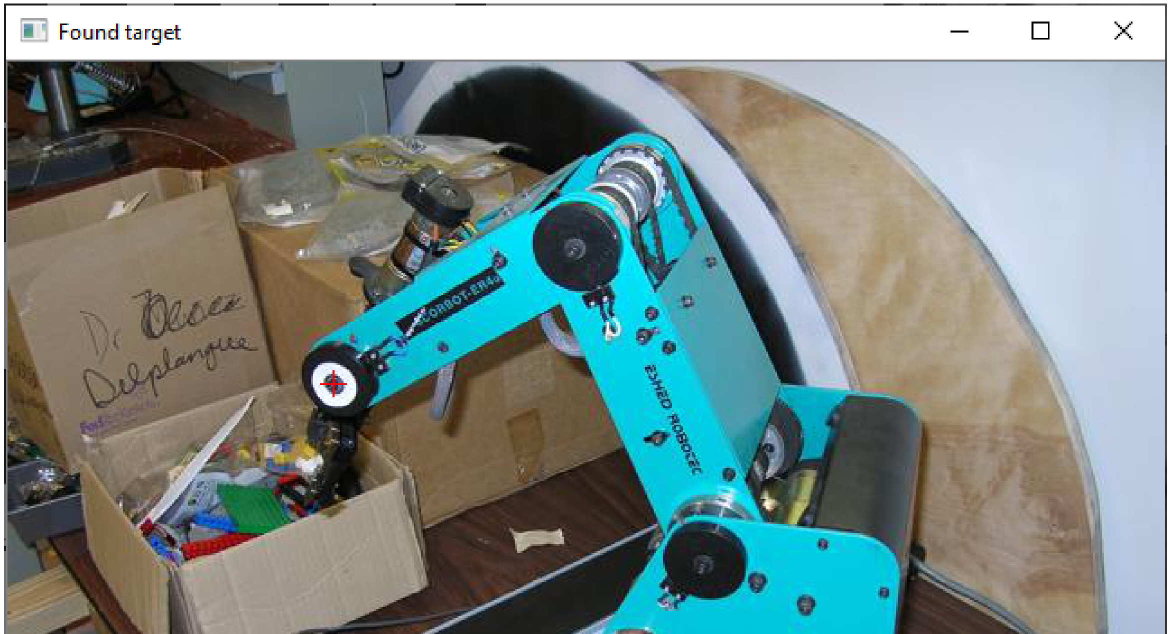
- Also, sometimes adaptive thresholding is better than global thresholding; especially when the lighting varies across the image

Overall strategy for finding CCC targets...

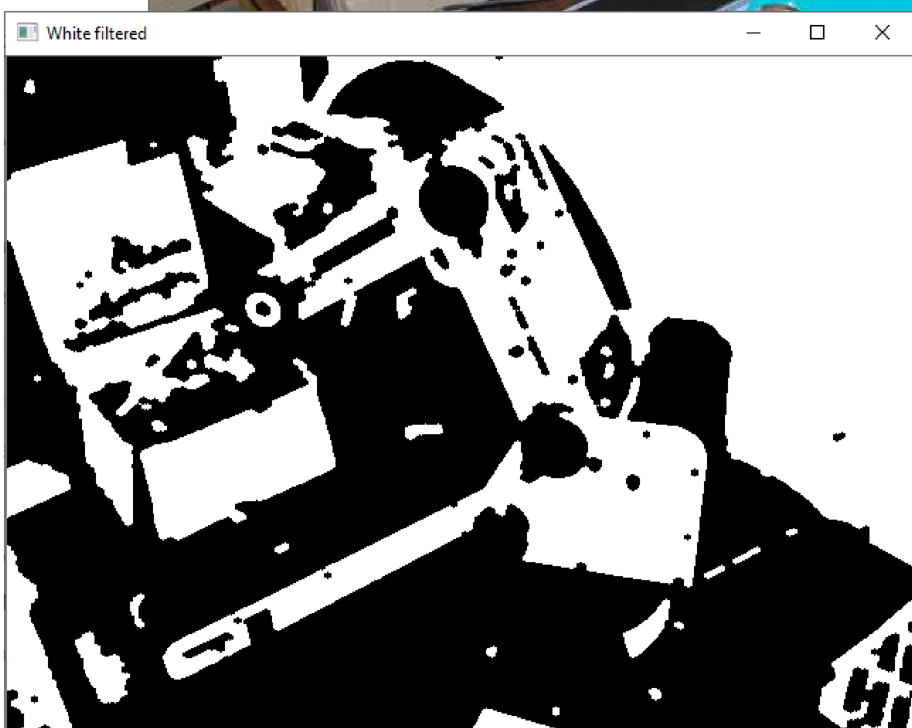
- Threshold the image
- Clean up the image using opening and closing
- Find connected components
- Find white blobs
- Complement the image and repeat the process to find black blobs
- Check every possible pair of white and black blobs…

```
for each white blob
  for each black blob
    Get centroid of white blob
    Get centroid of black blob
    if distance between centroids < thresh
      (Potentially also check if one bounding box encloses the other)
      We have a possible CCC! draw a crosshair at its centroid,
        or draw a rectangle around its bounding box
    end if
  end for
end for
```

Example



Here, I used a threshold of 3.0 pixels to check if centroids coincide



Fun fact – circle targets in space!