Image Segmentation

Dr. Su Su Maung
CEIT Department
Yangon Technological University

Outline of Lecture6

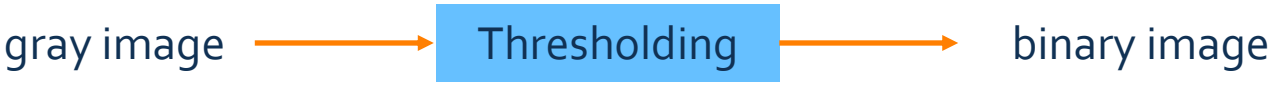
- Introduction
- Thresholding
- Edge Detection

Introduction

- What is segmentation?
- Segmentation refers to the operation of partitioning an image into component parts, or into separate objects.
  - thresholding, and
  - edge detection.

Thresholding

- Thresholding is a process of converting a grayscale input image to a binary image by using an optimal threshold.
  - Single Thresholding
  - Double Thresholding
  - Adaptive Thresholding



Single thresholding

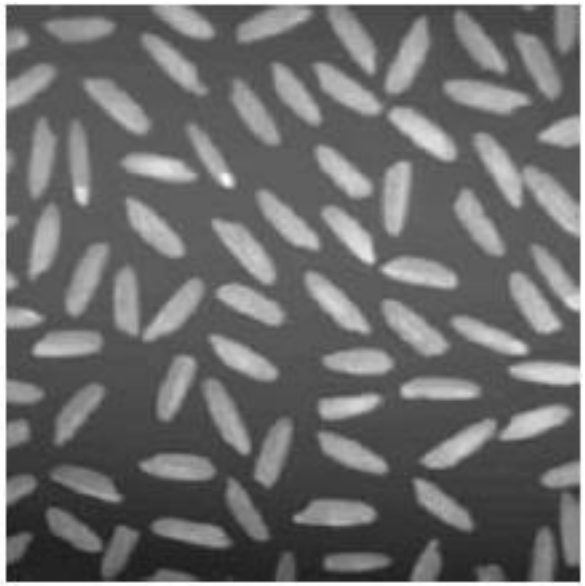
- Thresholding is the simplest method of image segmentation.
- A greyscale image is turned into a binary (black and white) image by first choosing a grey level *T* in the original image,

$$A\ pixel\ becomes \begin{cases} white\ if\ its\ grey\ level\ is > T \\ blaxk\ if\ its\ grey\ level\ is\ \leq T \end{cases}$$

- Thresholding is a vital part of image segmentation, where we wish to isolate objects from the background. It is also an important component of robot vision.
- For 8 bit image, stored as the variable X. Then the command `X>T`

```
f = io.imread('rice.jpg')
Io.imshow(f)
fig = plt.figue(); fig.show(io.imshow(f<50))
```

Single thresholding (cont.)



Grey image

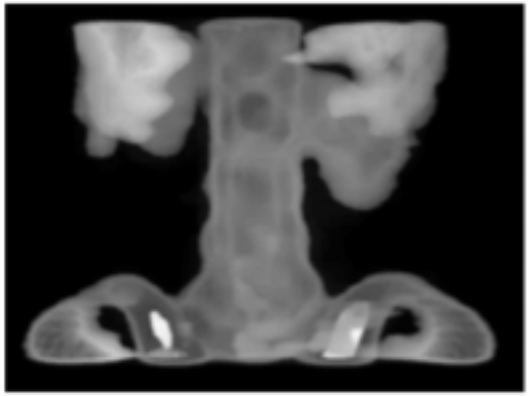
After thresholding

The resulting image can then be further processed to find the number, or average size of the grains.

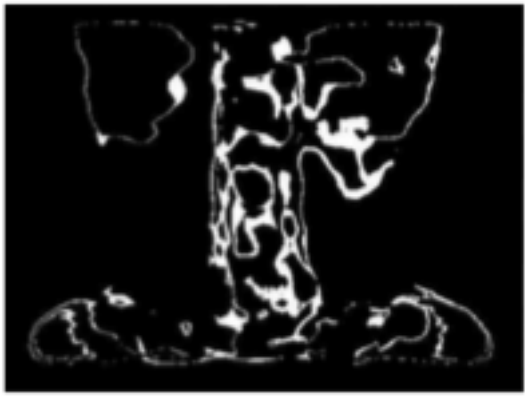
Double thresholding

- Here we choose two values T_1 and T_2 and apply a thresholding operation as: $X > T_1 \,\&\, X < T_2$

$$A\ pixel\ becomes \begin{cases} white\ if\ its\ grey\ level\ is\ between\ T_i\ and\ T_2, \\ black\ if\ its\ grey\ level\ is\ otherwise. \end{cases}$$



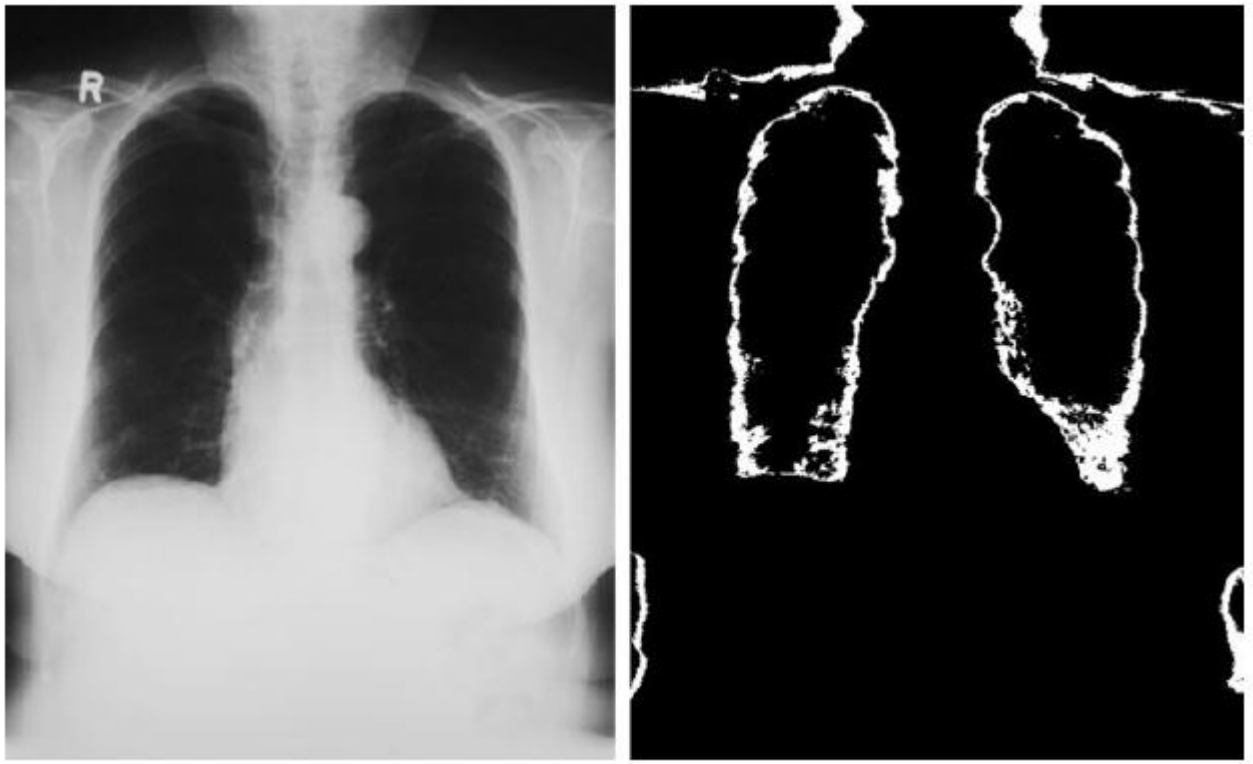
Grey image



Binary image after double thresholding

```
io.imshow((x>40) & (x<80))
```

Double thresholding

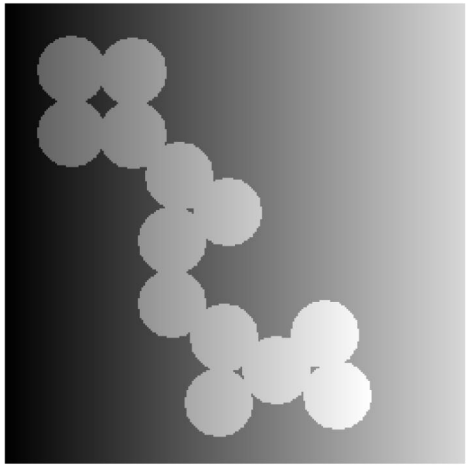


The image xray.png and the result after double thresholding

Note how double thresholding isolates the boundaries of the lungs, which single thresholding would be unable to do.

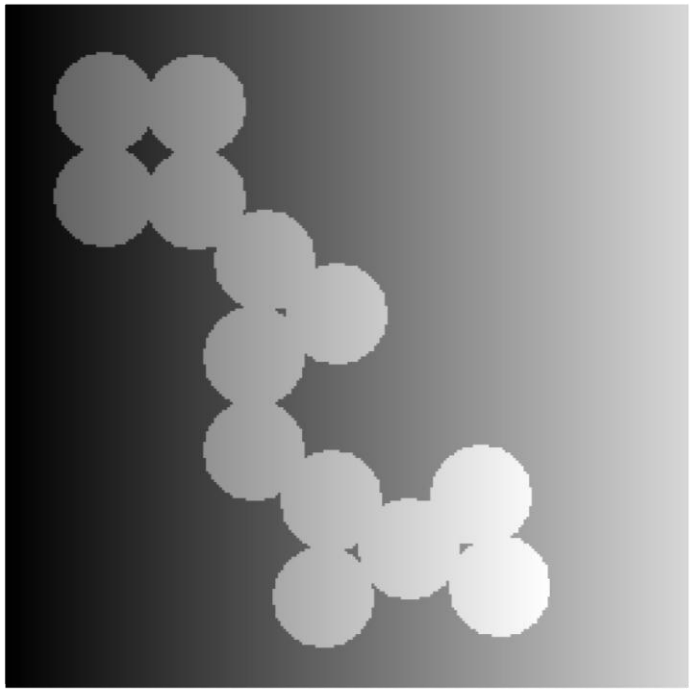
Adaptive thresholding

- Sometimes it is not possible to obtain a single threshold value which will isolate an object completely.
- This may happen if both the object and its background vary.

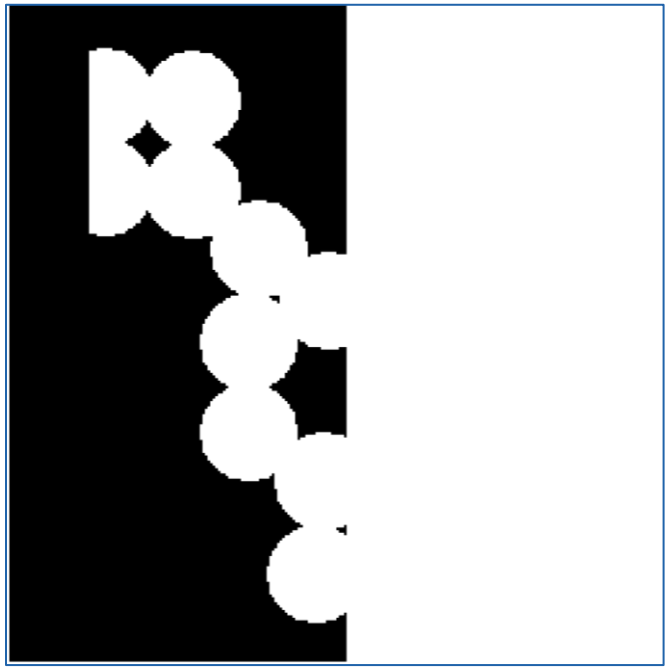


```
p = io.imread('circles.jpg').astype(float)
x,y = np.mgrid[0:r, 0:c].astype(float)
P2 = 255.0-p+y/2
io.imshow(p2)
```

Adaptive thresholding (cont.)

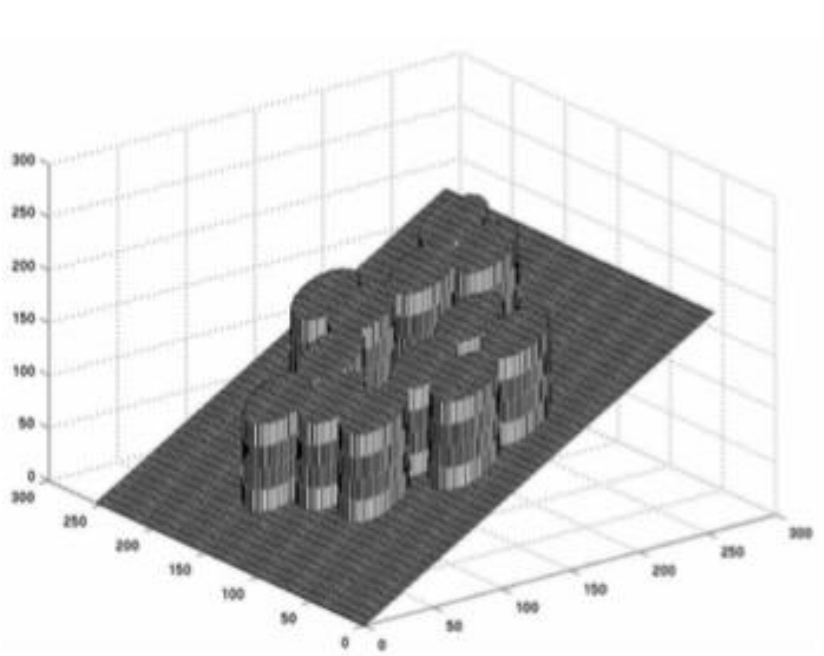


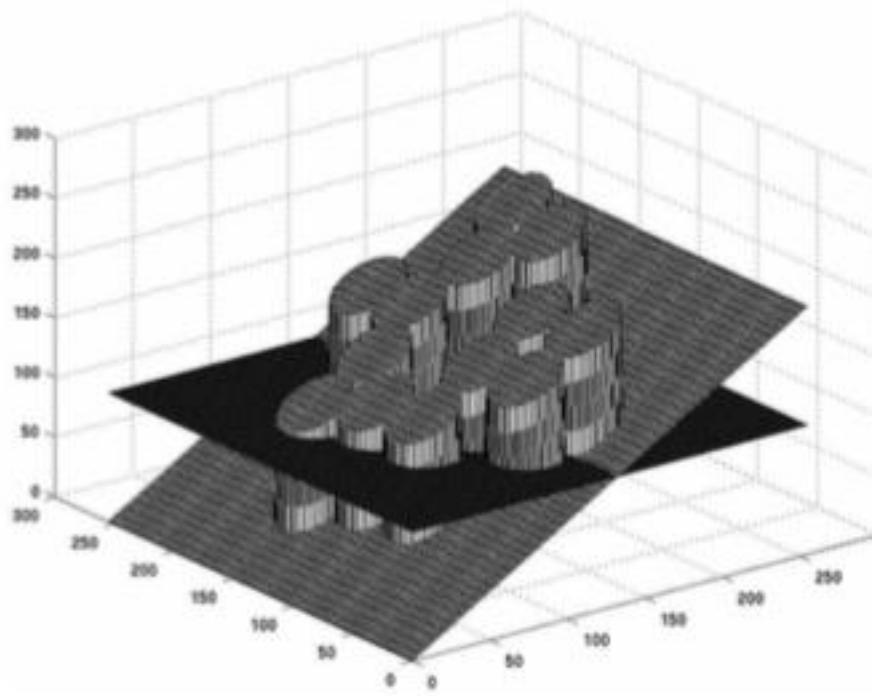
Circles image



Thresholding attempt

Adaptive thresholding (cont.)





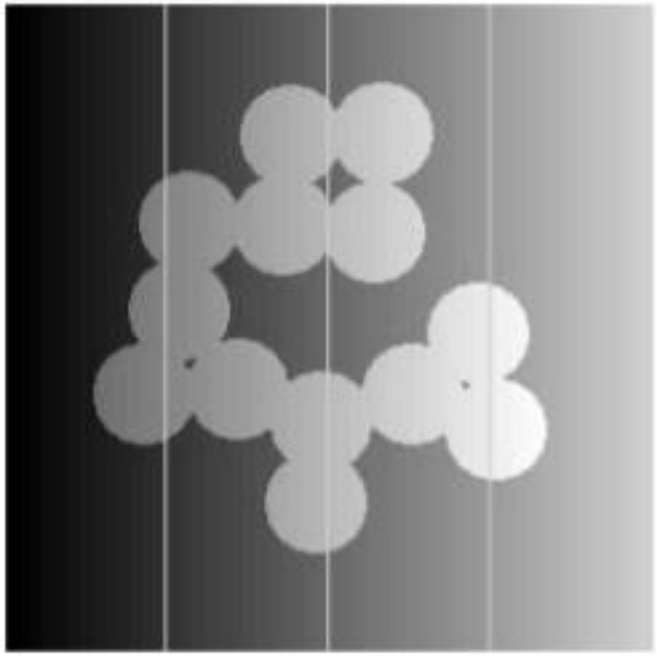
the threshold is shown on the right as a horizontal plane. It can be seen that no position of the plane can cut off the circles from the background.

Adaptive thresholding (cont.)

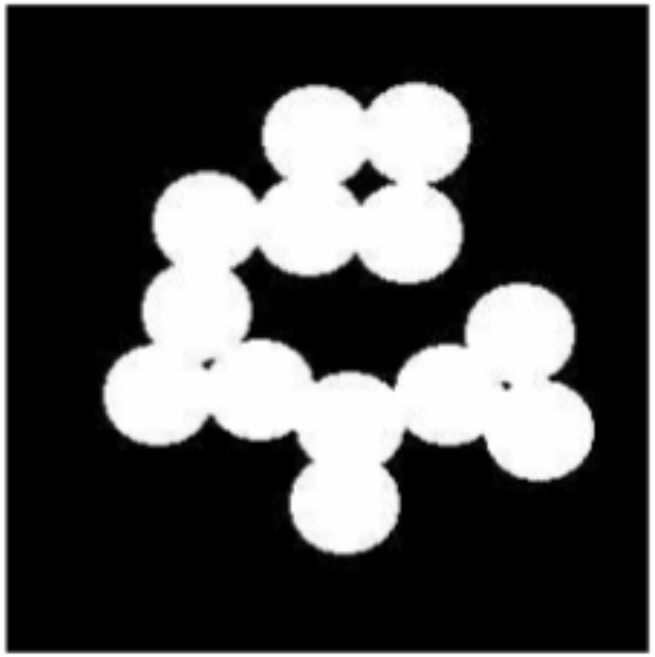
- What can be done in a situation like this is to cut the image into small pieces, and apply thresholding to each piece individually.
- Since in this particular example the brightness changes from left to right, we shall cut up the image into pieces and apply a threshold to each piece individually.
- The vectors starts and ends contain the column indices of the start and end of each block.

```
starts = range(0,c-1,162)
ends = range(162,c+1,162)
z = np.zero((r,c))
for i in range(6):
  temp = p2[:,starts[i]:ends[i]]
  z[:,starts[i]:ends[i]]=(temp>fl.threshold_otsu(temp))*1.0
```

Adaptive thresholding (cont.)



Cutting up the image



Thresholding each part separately

Cut up the image into four pieces. Threshold each piece. Display them as a single image

Applications of Thresholding

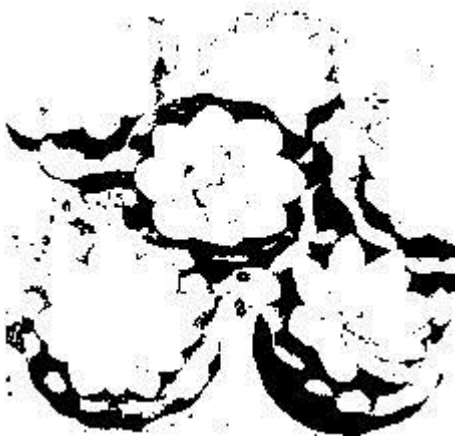
- We have seen that thresholding can be useful in the following situations:
  - When we want to remove unnecessary detail from an image, to concentrate on essentials by removing all gray level information, reduced to binary objects. But this information may be all we need to investigate sizes, shapes, or numbers of objects.
  - To bring out hidden detail obscured because of the similarity of the gray levels involved.
  - When we want to remove a varying background from an image. But the background in fact varies considerably. Thresholding at an appropriate value completely removes the background to show just the objects.

Choosing an Appropriate Threshold Value

- the success of segmentation depends very much on choosing an appropriate threshold level.
- If we choose a value too low, we may decrease the size of some of the objects, or reduce their number.
- Conversely, if we choose a value too high, we may begin to include extraneous background material.

Choosing an Appropriate Threshold Value

Original image



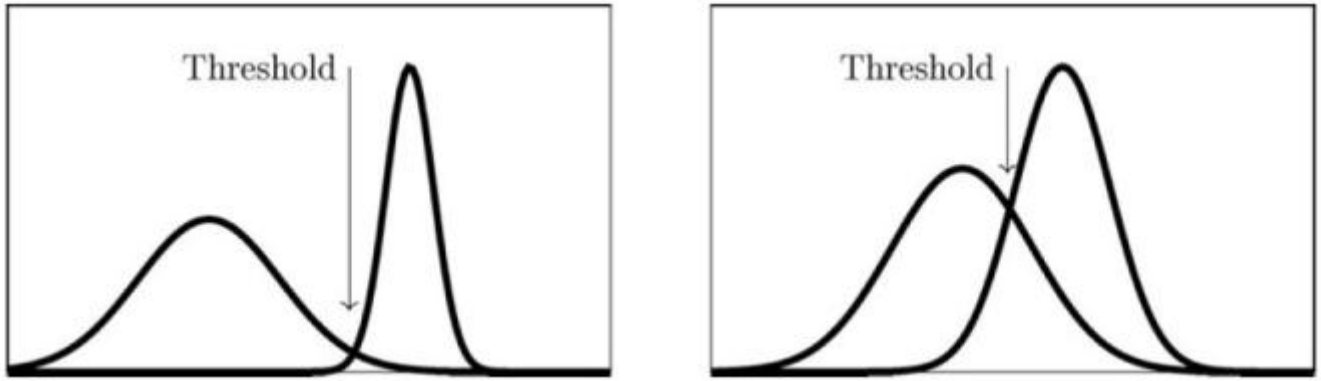
Threshold too low

Threshold too high

Attempts at different thresholding

Choosing an Appropriate Threshold Value

- The trouble is that in general the individual histograms of the objects and background will overlap, and
- without prior knowledge of the individual histograms it may be difficult to find a splitting point.
- assuming in each case that the histograms for the objects and backgrounds are those of a normal distribution. Then we choose the threshold values to be the place where the two histograms cross over.



Splitting up a histogram for thresholding

Choosing an Appropriate Threshold Value

- In practice, the histograms won't be as clearly defined as those in Figure.
- so we need some sort of automatic method for choosing a "best" threshold. There are in fact many different methods; here are two.
  - Otsu's Method
  - The ISODATA Method

Otsu's Method

- This was first described by Nobuyuki Otsu in 1979. It finds the best place to threshold the image into "foreground" and "background" components so that the *inter-class variance*, also known as the *between class variance*, is maximized.
- Suppose the image is divided into foreground and background pixels at a threshold t, and the fractions of each are ω_f and ω_b respectively.
- Suppose also that the means are μ_f and μ_b . Then the inter-class variance is defined as

$$\omega_f \omega_b (\mu_f - \mu_b)^2$$

Otsu's Method (cont.)

- If an image has *n* pixels of gray value *i*, then define *pi* = *ni* /*N* where *N* is the total number of pixels and L is the number of gray value in image. Thus, *pi* is the probability of a pixel having gray value *i*. Given a threshold value *t* then

$$\omega_b = \sum_{k=0}^{t-1} p_k$$

$$\omega_f = \sum_{k=t}^{L-1} p_k$$

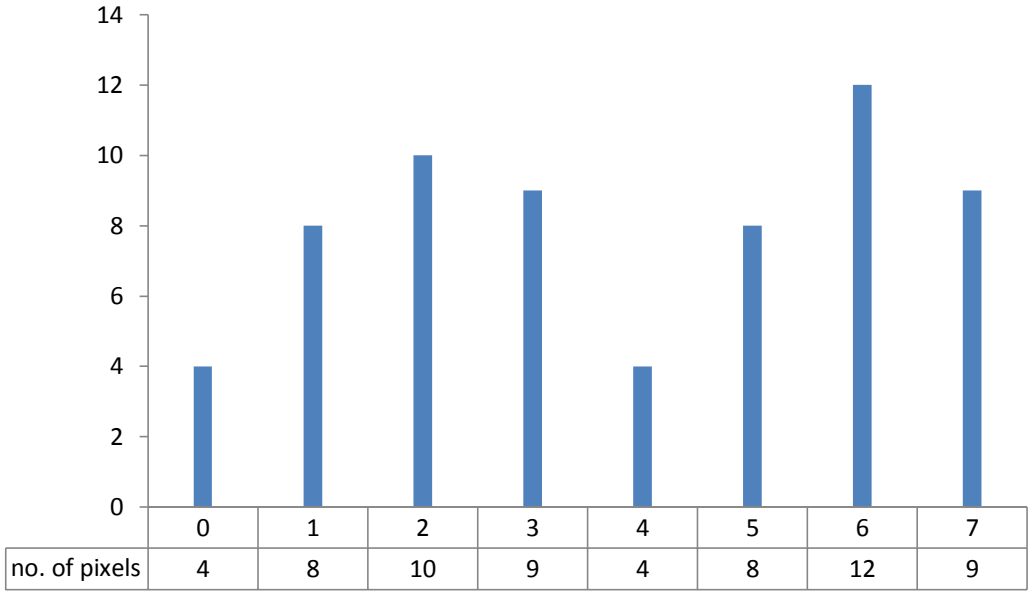
- The means can be defined as:

$$\mu_b = \frac{1}{\omega_b} \sum_{k=0}^{t-1} k p_k$$

$$\mu_f = \frac{1}{\omega_f} \sum_{k=t}^{L-1} k p_k$$

Otsu's Method (cont.)

i	0	1	2	3	4	5	6	7
n_i	4	8	10	9	4	8	12	9
p_i	0.0625	0.125	0.15625	0.140025	0.0625	0.125	0.1875	0.140625



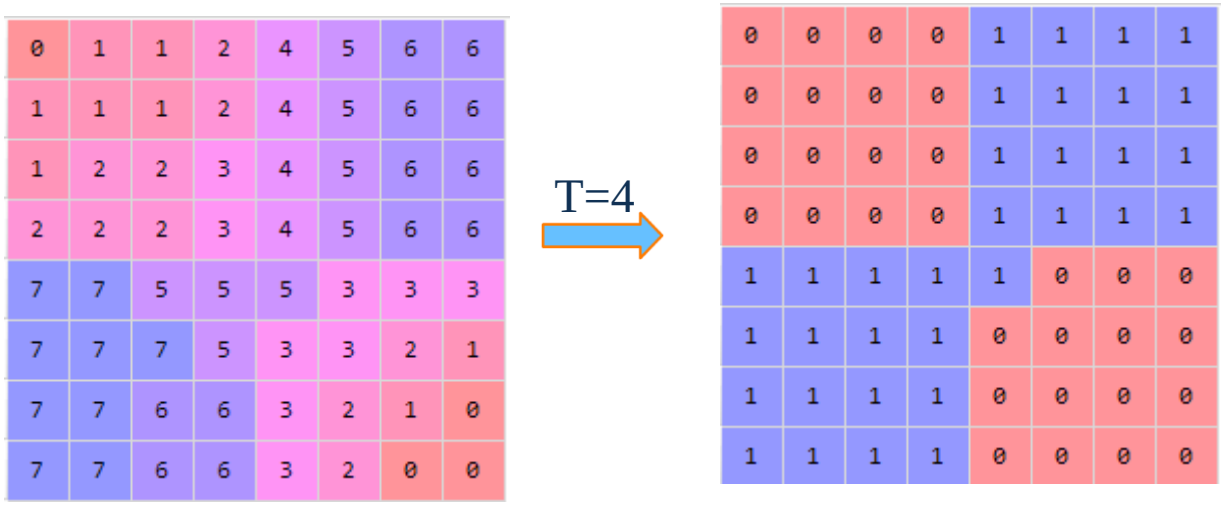
An example of a histogram illustrating Otsu's method

Otsu's Method (cont.)

k	n_k	p_k	ω_b	ω_f	kp_k	$\sum kp_k$	μ_b	μ_f	vars
0	4	0.0625	0.0625	0.9375	0	0	0	4.1	0.98496
1	8	0.125	0.1875	0.8125	0.125	0.125	0.66667	4.57692	2.32935
2	10	0.15625	0.34375	0.65625	0.3125	0.4375	1.27273	5.19048	3.46246
3	9	0.14062	0.48438	0.51562	0.42188	0.85938	1.77419	5.78788	4.02348
4	4	0.0625	0.54688	0.45312	0.25	1.10938	2.02857	6.03448	3.97657
5	8	0.125	0.67188	0.32812	0.625	1.73438	2.58140	6.42857	3.26296
6	12	0.1875	0.85938	0.14062	1.125	2.85938	3.32727	7	1.63013
7	9	0.14062	1	0	0.98438	3.84375	3.84375	-	-

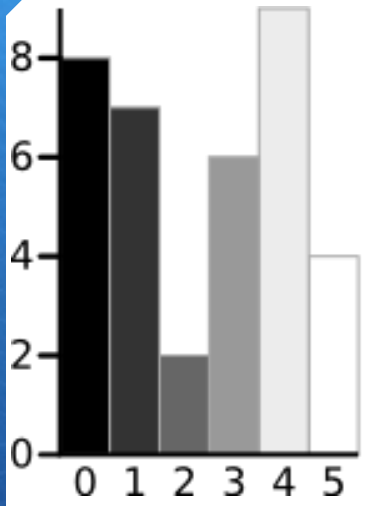
The largest value of the inter-class variance is at $k = 3$, which means that $t - 1 = 3$ or $t = 4$ is the optimum threshold.

Otsu's Method (cont.)



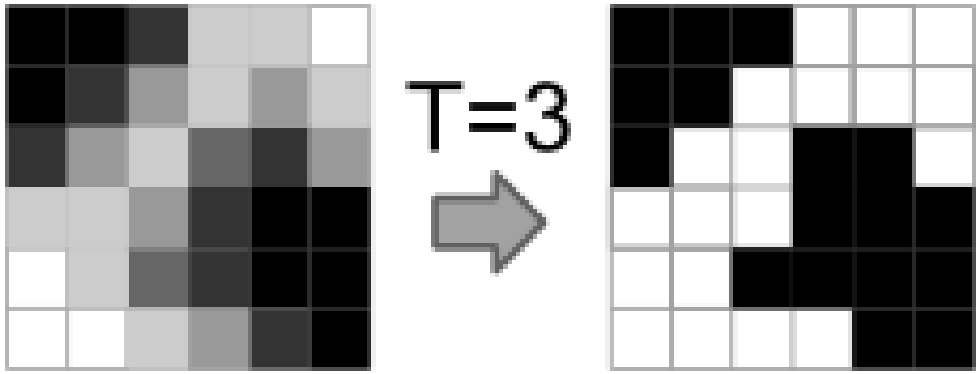
Input image and output image after Otsu's thresholding

Otsu's Method



0	0	1	4	4	5
0	1	3	4	3	4
1	3	4	3	2	3
4	4	3	1	1	0
5	4	2	1	1	0
5	5	4	3	2	0

A 6-level greyscale image and its histogram



Result of Otsu's Method

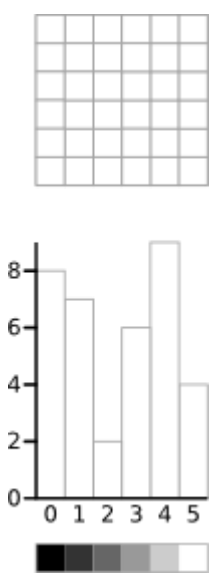
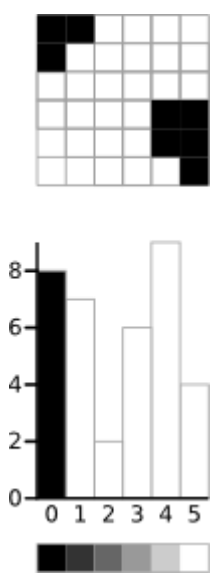
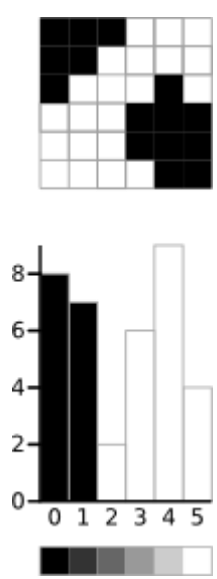
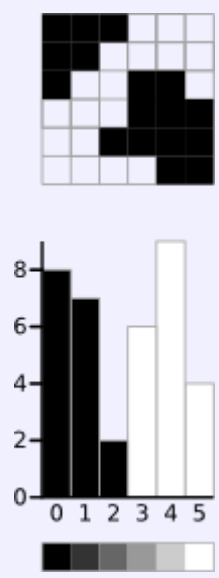
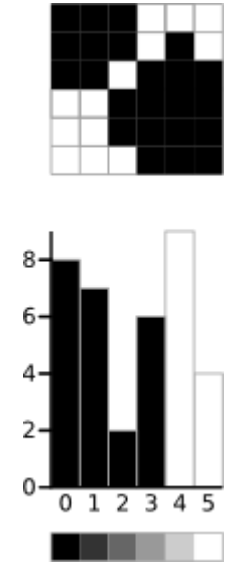
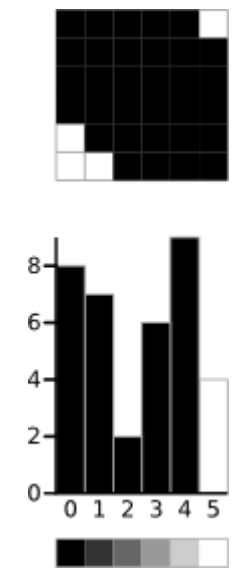
0	0	1	4	4	5
0	1	3	4	3	4
1	3	4	2	1	3
4	4	3	1	0	0
5	4	2	1	0	0
5	5	4	3	1	0

T=3

0	0	0	1	1	1
0	0	1	1	1	1
0	1	1	0	0	1
1	1	1	0	0	0
1	1	0	0	0	0
1	1	1	1	0	0

Result of Otsu's Method by Python

Otsu's Method

Threshold	T=0	T=1	T=2	T=3	T=4	T=5
						

The ISODATA Method

- ISODATA is an acronym for "Iterative Self-Organizing Data Analysis Technique A," where the final "A" was added, according to the original paper, "…to make ISODATA pronouncable."

- It is in fact a very simple method, and in practice converges very fast:

1. Pick a precision value ε and a starting threshold value t. Often $t = L/2$ is used.
2. Compute μ_f and μ_b for this threshold value.
3. Compute $t' = (\mu_b + \mu_f)/2$
4. If $|t - t'| < \varepsilon$ the stop and return t, else put $t = t'$ and go back step 2.

The ISODATA Method

```
k = np.arange(256)
n = ndi.histogram(c,0,255,256);p = n/ (c.size + 0.0)
wb = np.cumsum(k*p);wf = 1 – wb
Kpc = np.cumsum(k*p)
mu_b = kpc/wb
mu_f = (kpc[-1]-kpc)/wf
```

```
t = 128
for i in range(10):
   t1 = int ((mu_f[t]+mu_b[t])/2.0)
    print t1
    t = t1
108
95
90
88
88
88
```

The iteration quickly converges in only four steps.

The ISODATA Method (cont.)

- In Python there are the methods threshold_otsu and threshold_isodata in the filters module of skimage.
- Note that Python returns an 8-bit integer threshold value if that is the initial image type.
- Once the optimum threshold value has been obtained, it can be applied to the image.

Assignment

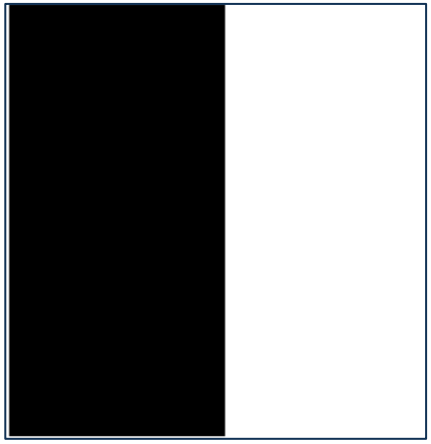
Find the best place to threshold the images into foreground and background components using Otsu's method.

2	3	6	0	0	0	1	1
1	1	3	0	0	0	1	1
3	2	4	2	5	5	5	7
3	3	5	4	4	5	6	7
5	5	3	2	4	2	1	1
5	3	3	4	2	1	1	1
5	3	2	2	4	6	7	7
5	3	4	4	6	6	7	7

Edge Detection

Edge

- Abrupt change in the intensity of pixels
- Discontinuity in image brightness or contrast
- Usually edges occur on the boundary of two regions



Edge in pixels

- An edge may be loosely defined as a local discontinuity in the pixel values which exceeds a given threshold.
- More informally, an edge is an observable difference in pixel values.

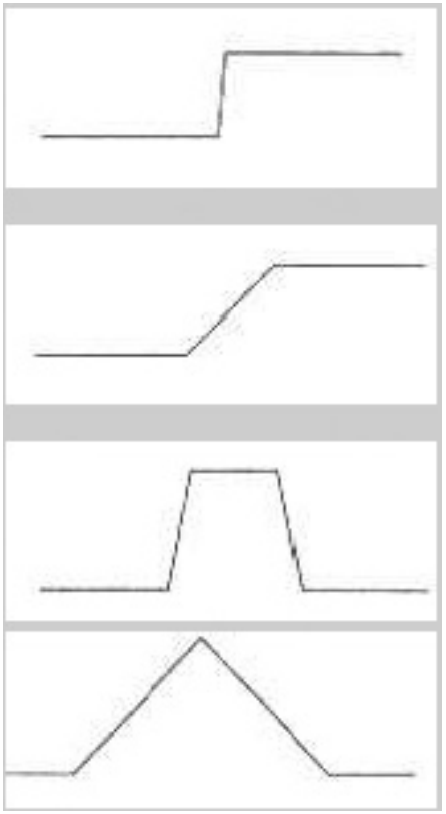
51	52	53	59
54	52	53	62
50	52	53	68
55	52	53	55

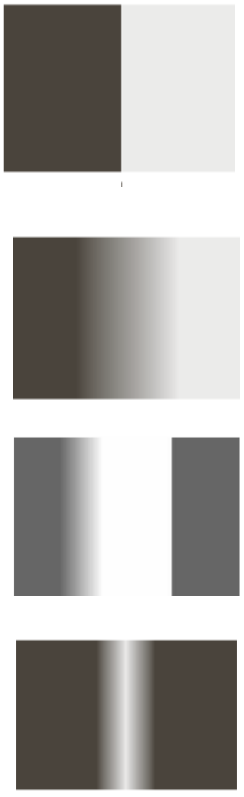
50	53	155	160
51	53	160	170
52	53	167	190
51	53	162	155

Blocks of pixels

Edge Types

- Step edge (ideal edge)
  - Gray values change suddenly
- Ramp edge
  - Gray values change slowly
- Ridge
- Roof





Edge detection

- Edges contain some of the most useful information in an image.
- We may use edges
  - to measure the size of objects in an image;
  - to isolate particular objects from their background;
  - to recognize or classify objects.

In Python, edge detection methods are in the filters module of skimage .

Methods of Edge Detection

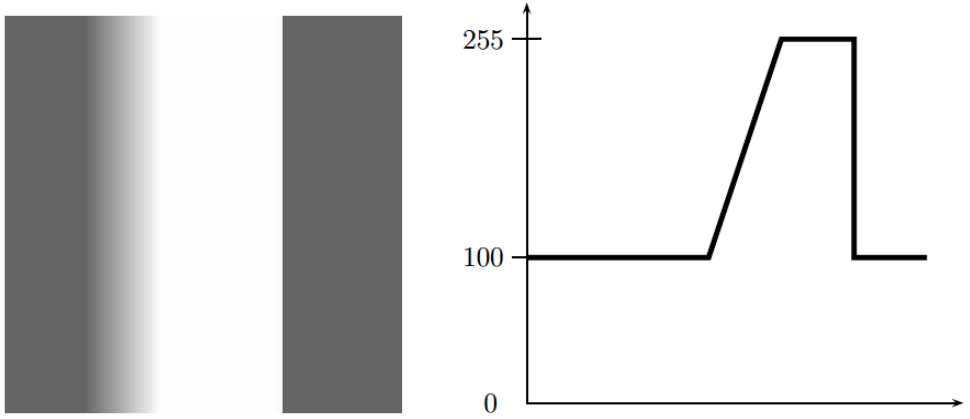
- Gradient methods (First Order Derivative)
  - Compute Gradient magnitude horizontally and vertically
- Zero-crossing methods (Second Order Derivative)
  - Laplacian of an image
  - Locate zeros in the second derivative of an image

Gradient based Edge Detection

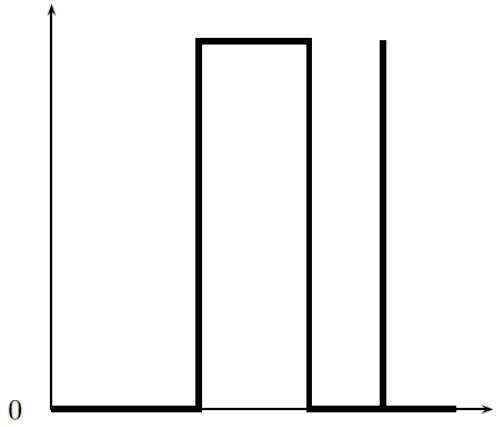
- Roberts Edge Detector
- Prewitt Edge Detector
- Sobel Edge Detector
- Canny Edge Detector

Derivatives and edges

- we plot the gray values as we traverse the image from left to right.
- Two types of edges are illustrated here: a ramp edge, where the grey values change slowly, and a step edge, or an ideal edge, where the grey values change suddenly.



Edges and their profiles



The derivative of the edge profile

Derivatives and edges (cont.)

- Many edge finding operators are based on differentiation; to apply the continuous derivative to a discrete image, first recall the definition of the derivative:

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \qquad \lim_{h \to 0} \frac{f(x) - f(x-h)}{h}, \qquad \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}$$

- a discrete version of the derivative expression is

$$f(x+1) - f(x). \qquad f(x) - f(x-1), \qquad (f(x+1) - f(x-1))/2.$$

- For an image, with two dimensions, we use partial derivatives; gradient,

$$\left[\frac{\partial f}{\partial x} \quad \frac{\partial f}{\partial y}\right]$$

- With direction and magnitude

$$\tan^{-1}\left(\frac{\partial f/\partial y}{\partial f/\partial x}\right) \qquad \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}.$$

Most edge detection methods are concerned with finding the magnitude of the gradient, and then applying a threshold to the result.

Prewitt Edge Detector (Edge detection filters)

- Using the expression $f(x+1) - f(x-1)$ for the derivative, leaving the scaling factor out, produces horizontal and vertical filters:

$$\begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$$

- These filters will find vertical and horizontal edges in an image and produce a reasonably bright result. However, the edges in the result can be a bit "jerky"; this can be overcome by smoothing the result in the opposite direction; by using the filters

$$\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$$

Prewitt Edge Detector (cont.)

- Both filters can be applied at once, using the combined filter:

$$P_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

Finding vertical edges

$$P_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

Finding horizontal edges

- These filters are the Prewitt filters for edge detection.
- If p_x and p_y are the grey values produced by applying filter to an image, then the magnitude of the gradient is obtained with

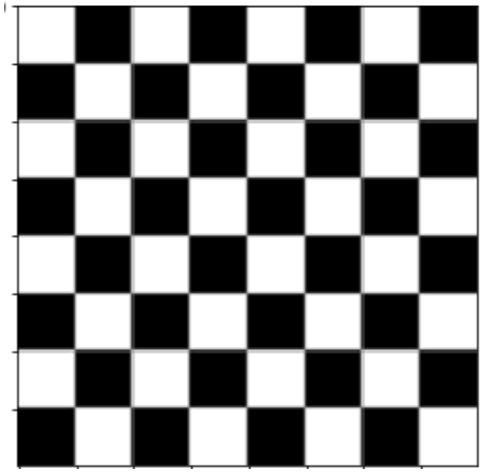
$$\sqrt{p_x^2 + p_y^2}. \qquad \max\{|p_x|, |p_y|\} \qquad |p_x| + |p_y|.$$

Prewitt Edge Detector (cont.)

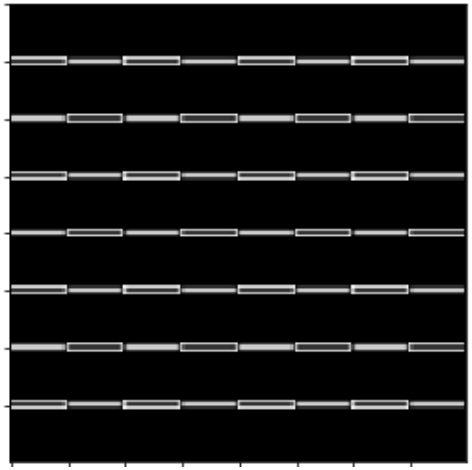
```
import skimage.data as data
img=data.checkerboard()
```

```
imgx=fl.prewitt_h(img)
plt.figure()
io.imshow(np.uint8(imgx*255))
```

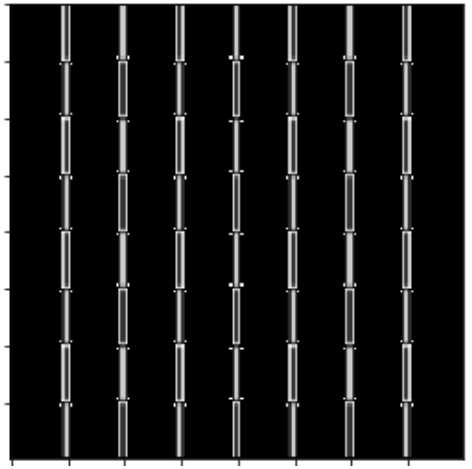
```
imgy=fl.prewitt_v(img)
plt.figure()
io.imshow(np.uint8(imgy*255))
```



Original image



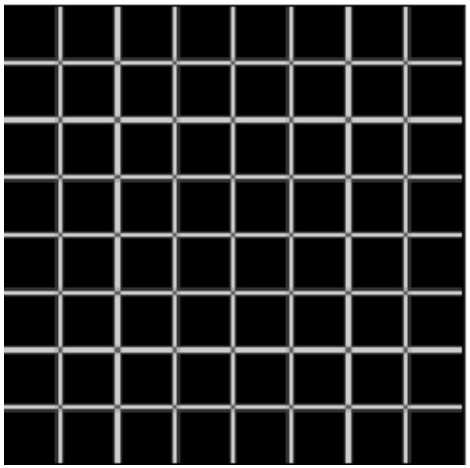
Horizontal Edge



Vertical Edge

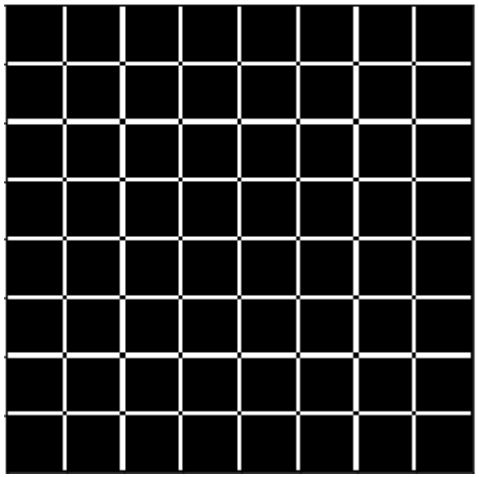
Prewitt Edge Detector (cont.)

```
edge_p=np.sqrt(imgx**2+imgy**2)
```



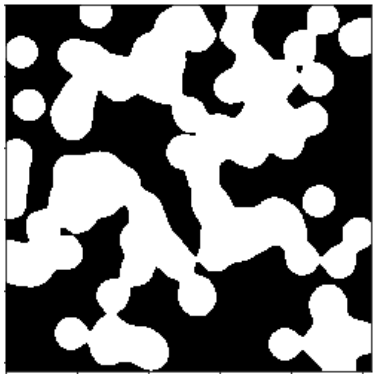
Edge image after finding magnitude

```
edge_pb=edge_p>fl.threshold_otsu(edge_p)
```

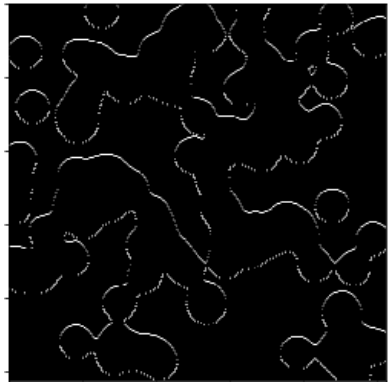


Binary image after Outsu's thresholding

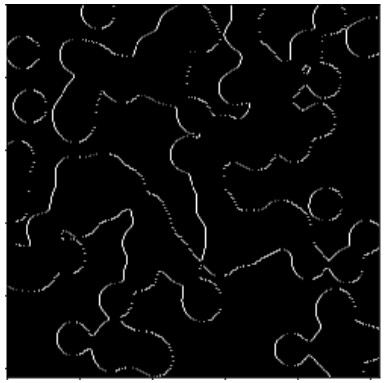
Prewitt Edge Detector (cont.)



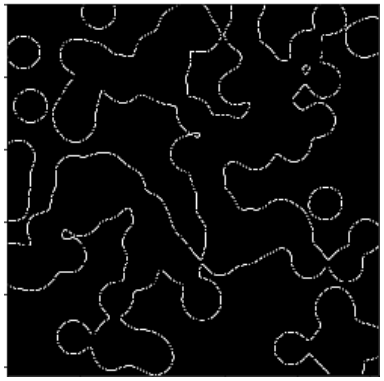
Binary blobs image



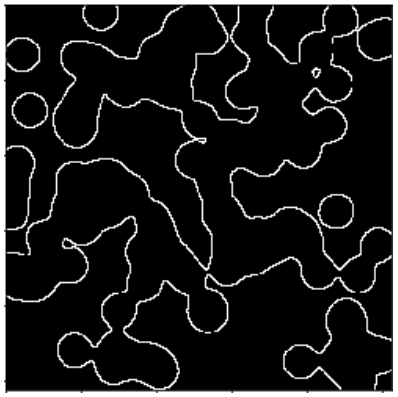
Horizontal edge



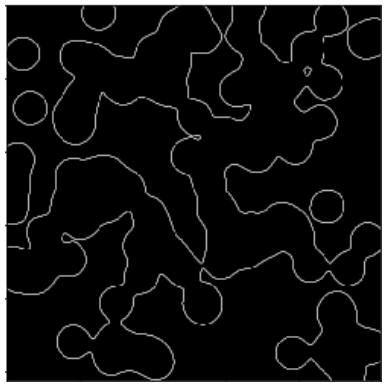
Vertical edge



Gray edge image



Binary edge image



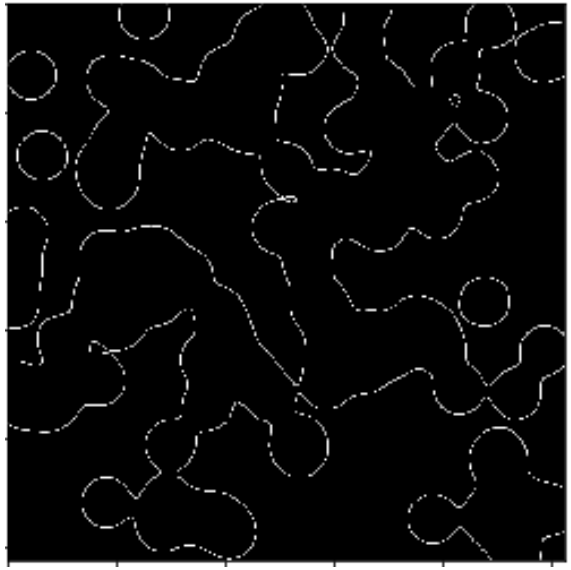
Binary image using fl.prewitt()

Roberts filter

Roberts filters:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \text{ and } \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

```
edge_r=fl.roberts(img)
plt.figure()
io.imshow(np.uint8(edge_r*255))
```



Roberts edge detection

Sobel filters

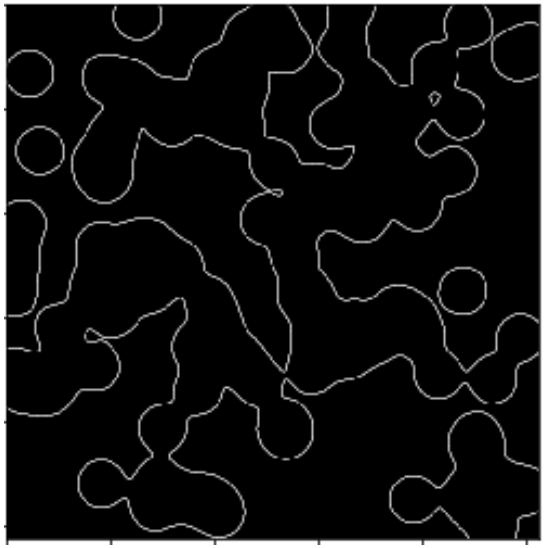
Sobel filters:

- The *Sobel filters* are similar to the Prewitt filters in that they apply a smoothing filter in the opposite direction to the central difference filter.

- In the Sobel filters, the smoothing takes the form which $\begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \text{ and } \begin{bmatrix} -1 & -2 & 1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

```
edge_s=fl.sobel(img)
plt.figure()
io.imshow(np.uint8(edge_s*255))
```



Sobel edge detection

Second derivatives

- The Laplacian
- Another class of edge-detection method is obtained by considering the second derivatives. The sum of second derivatives in both directions is called the laplacian;

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}.$$

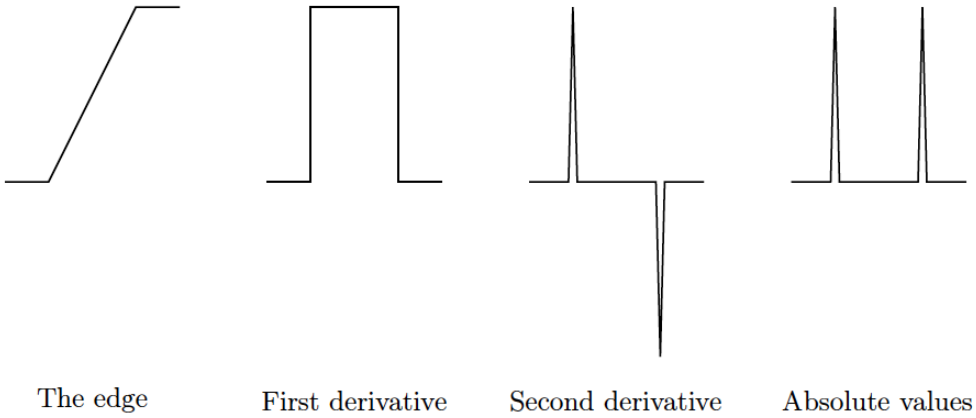
- it can be implemented by the filter

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

- This is known as a discrete Laplacian.

The Laplacian

- The laplacian has the advantage over first derivative methods in that it is an isotropic filter; this means it is invariant under rotation.
- However, a major problem with all second derivative filters is that they are very sensitive to noise.



- The Laplacian (after taking an absolute value, or squaring) gives double edges.

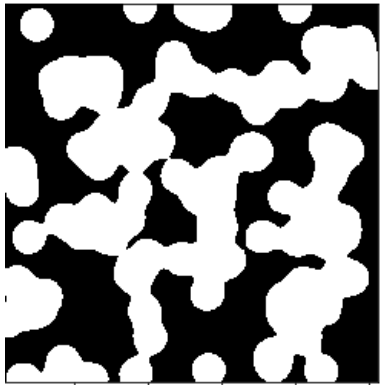
The Laplacian (cont.)

- Python commands:

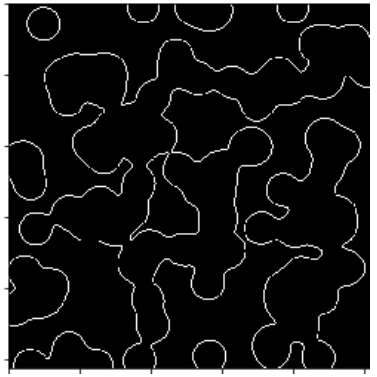
```
img=data.binary_blobs()
img2=ut.img_as_float(img)
img_la=np.abs(fl.laplace(img2))
```

- Laplacian masks:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \text{ and } \begin{bmatrix} -2 & 1 & -2 \\ 1 & 4 & 1 \\ -2 & 1 & -2 \end{bmatrix}$$



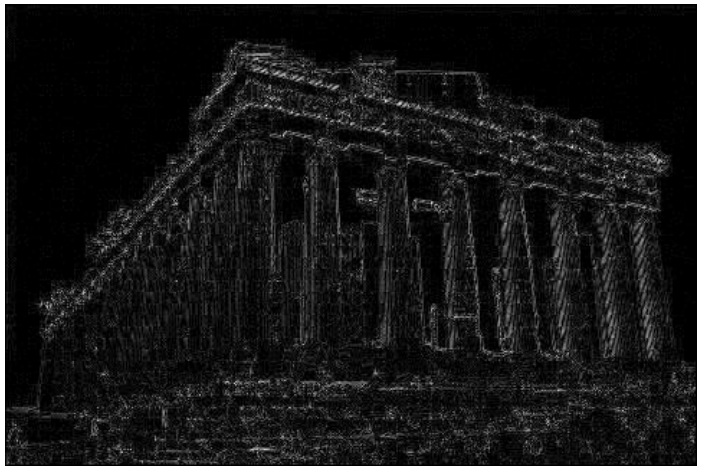
Binary blobs image



Edge with a discrete Laplacian

The Laplacian (cont.)

Original image



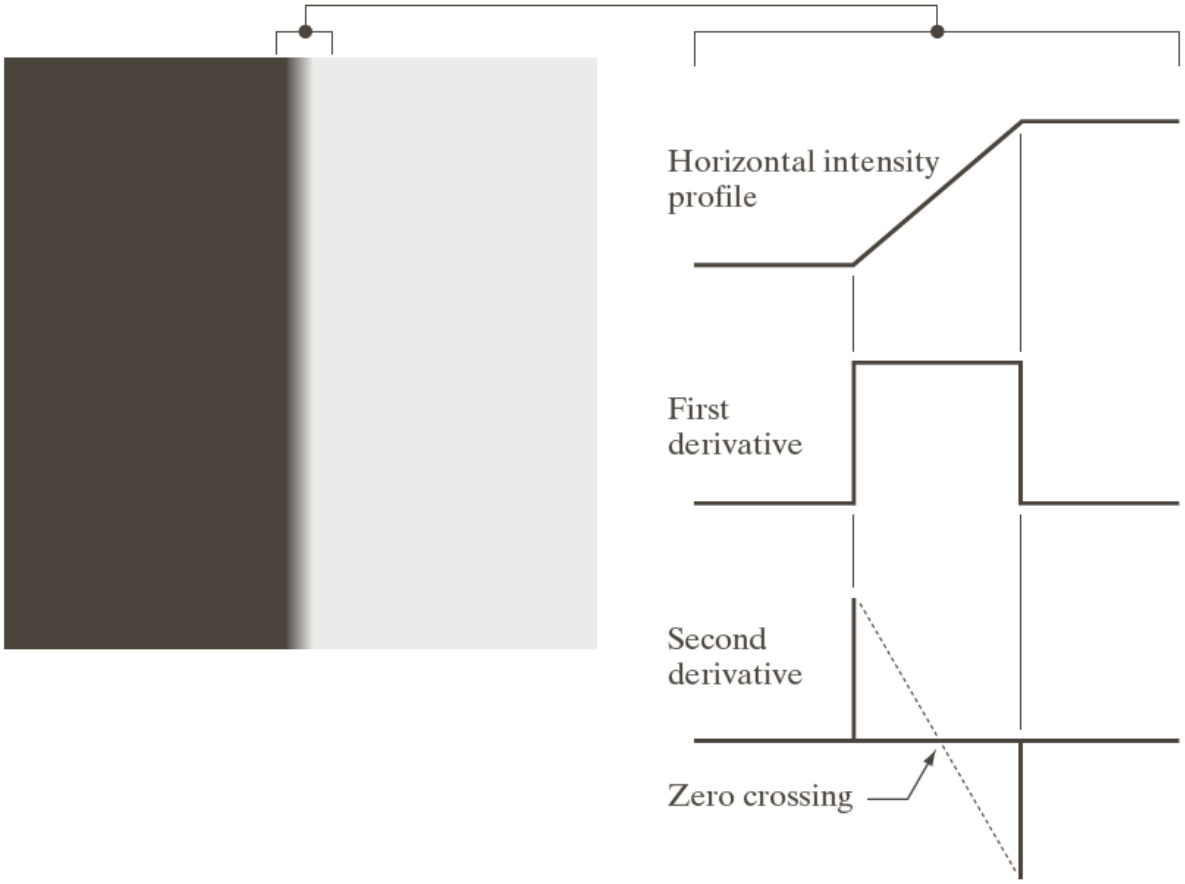
Edge with a discrete Laplacian

To use the Laplacian it is appropriat to find the position of edges by locating zero crossings

Zero crossings

- Position of edges can be found by looking for zero crossings after filtering the image with Laplacian filter.
- the position of the edge is given by the place where the value of the filter takes on a zero value.
- In general, these are places where the result of the filter changes sign.

Zero crossings (cont.)



Zero crossings (cont.)

- We define the zero crossings in such a filtered image to be pixels which satisfy either of the following:
  - they have a negative grey value and are next to (by four-adjacency) a pixel whose grey value is positive,
  - they have a value of zero, and are between negative and positive valued pixels.

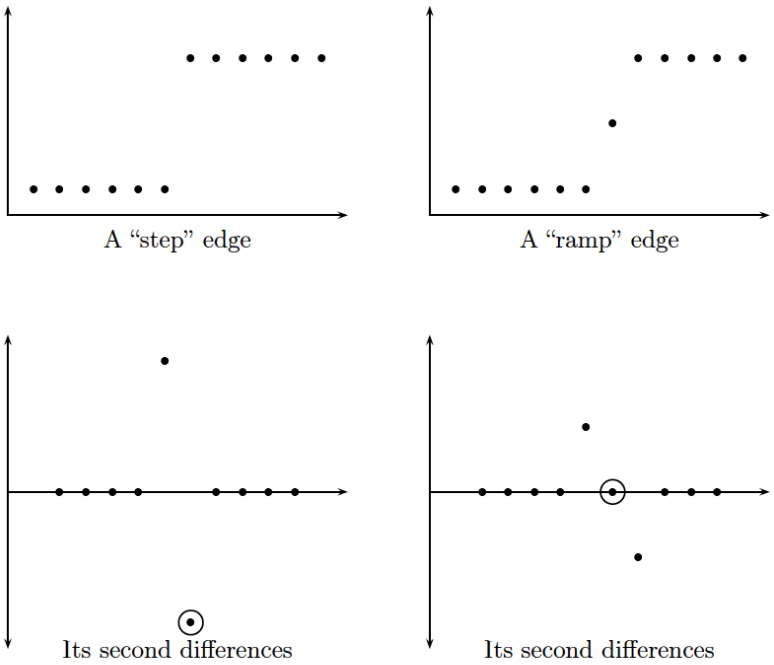


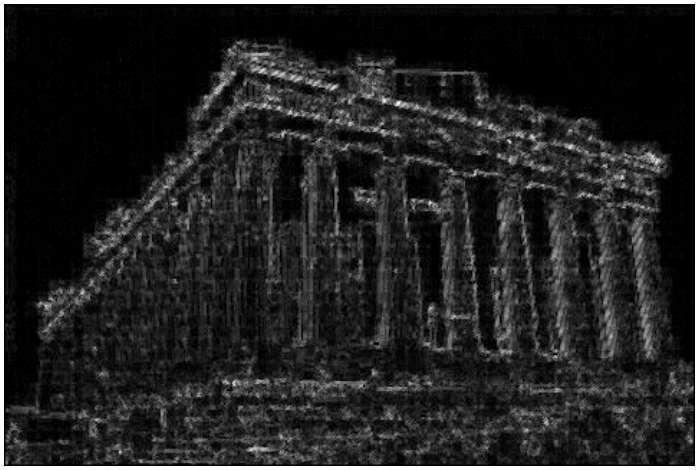
Figure 8.12: Edges and second differences

Zero crossings (cont.)

- take the zero-crossings after a laplace filtering.

```
img2 = ut.img_as_float(img)
img_lap = abs(fl.laplace(img2))
```

This is not in fact a very good result. Too many grey level changes have been interpreted as edges by this method.



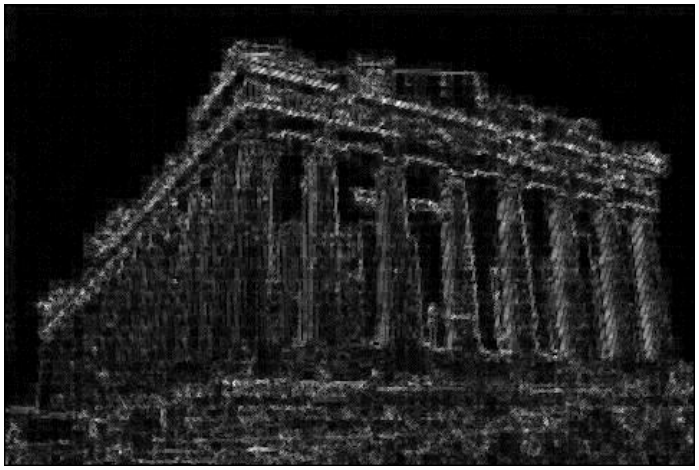
Zeros crossings

Zero crossings (cont.)

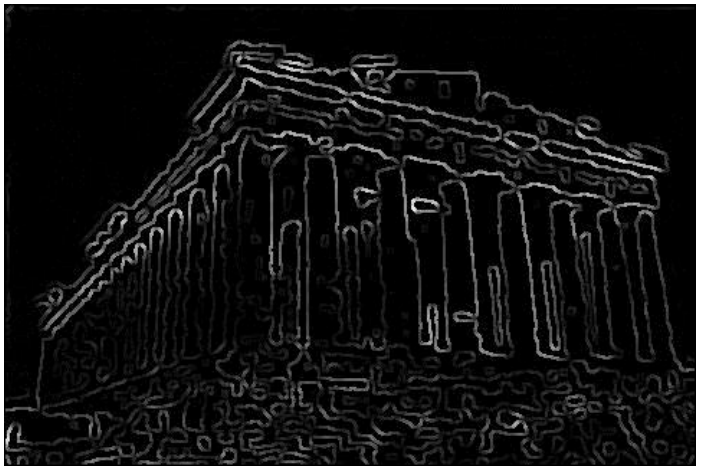
- To eliminate them, we may first smooth the image with a Gaussian filter.
  - smooth the image with a Gaussian filter,
  - convolve the result with a laplacian,
  - find the zero crossings.

```
img2 = ndi.gaussian_laplace(np.float64(img),3)
Img_edge = Zero_crossing(img2)
```

Zero crossings



Zeros crossings



Using an LoG filter first

Python code

```
def Zero_crossing(image):
    z_c_image = np.zeros(image.shape)
    for i in range(1, image.shape[0] - 1):
        for j in range(1, image.shape[1] - 1):
            negative_count = 0
            positive_count = 0
            neighbour = [image[i+1, j-1],image[i+1, j],image[i+1, j+1],image[i, j-1],image[i, j+1],image[i-1, j-1],image[i-1, j],image[i-1, j+1]]
            d = max(neighbour)
            e = min(neighbour)
            for h in neighbour:
                if h>0:
                    positive_count += 1
                elif h<0:
                    negative_count += 1
            z_c = ((negative_count > 0) and (positive_count > 0))
            if z_c:
                if image[i,j]>0:
                    z_c_image[i, j] = image[i,j] + np.abs(e)
                elif image[i,j]<0:
                    z_c_image[i, j] = np.abs(image[i,j]) + d
    z_c_norm = z_c_image/z_c_image.max()*255
    z_c_image = np.uint8(z_c_norm)
    return z_c_image
```

Python code (cont.)

```
#Otsu Method
 import numpy as np
img=np.array([[1,2,4,3,5,3,4,4],[4,4,2,3,3,5,6,6],[2,1,0,1,5,5,5,7],[2,2,5,
0,0,5,6,7],[0,0,2,1,3,1,6,6],[5,2,2,3,1,6,6,6],[5,2,1,1,3,6,7,7],[5,2,3,3,6,6,
7,7]])
l,w=img.shape
n=l*w
L=img.max()
bin=np.zeros(L+1)
binwb=np.zeros(L+1)
binnew=np.zeros(L+1)
meanb=np.zeros(L+1)
meanf=np.zeros(L+1)
var=np.zeros(L+1)
for i in range(l):
   for j in range(w):
      for k in range(L+1):
         if img[i][j]==k :
            bin[k]=bin[k]+1

binp=bin/n
for k in range(L+1):
   for p in range(k+1):
      binwb[k]+=binp[p]
binwf=1-binwb
```

```
for i in range(L+1):
   b=0
   a=0
   for j in range(i+1):
      a+=(j*bin[j])
      b+=bin[j]
   meanb[i]=a/b
for p in range(L+1):
   b=0
   a=0
   for q in range(p+1, L+1):
      a+=(q*bin[q])
      b+=bin[q]
   if b==0: break
   meanf[p]=a/b
var=(binwb*binwf)*(meanf-meanb)*(meanf-meanb)
value=var.max()
for i in range(L+1):
   if var[i]==value :
      thread=i+1
      break
imgnew=img>thread
```

Next Week Lecture (Week7)

- Lecture7:Mathematical Morphology

Thank You